Microkernel Architecture and Hardware Abstraction Layer of a Reliable Reconfigurable Real-Time Operating System (R3TOS)

Microkernel Architecture and Hardware Abstraction Layer of a Reliable Reconfigurable Real-Time Operating System (R3TOS)

XABIER ITURBE, University of Edinburgh and IK4-Ikerlan Research Alliance
KHALED BENKRID, University of Edinburgh
CHUAN HONG, University of Edinburgh
ALI EBRAHIM, University of Edinburgh
RAUL TORREGO, IK4-Ikerlan Research Alliance
TUGHRUL ARSLAN, University of Edinburgh

This article presents a new solution for easing the development of reconfigurable applications using Field-Programable Gate Arrays (FPGAs). Namely, our Reliable Reconfigurable Real-Time Operating System (R3TOS) provides OS-like support for partially reconfigurable FPGAs. Unlike related works, R3TOS is founded on the basis of resource reusability and computation ephemerality. It makes intensive use of reconfiguration at very fine FPGA granularity, keeping the logic resources used only while performing computation and releasing them as soon as it is completed. To achieve this goal, R3TOS goes beyond the traditional approach of using reconfigurable slots with fixed boundaries inter-connected by means of a static communication infrastructure. Instead, R3TOS approaches a static route free system where nearly everything is reconfigurable. The tasks are concatenated to form a computation chain through which partial results naturally flow, and data is exchanged among remotely located tasks using FPGA's reconfiguration mechanism or by means of "removable" routing circuits. In this article, we describe the R3TOS microkernel architecture as well as its hardware abstraction services and programming interface. Notably, the article presents a set of novel circuits and mechanisms to overcome the limitations and exploit the opportunities of Xilinx reconfigurable technology in the scope of hardware multitasking and dependability.

Categories and Subject Descriptors: C.1.4 [**Computer systems organization**]: Reconfigurable computing; B.3.6 [**Hardware**]: Reconfigurable logic and FPGAs

General Terms: Design, Theory

Additional Key Words and Phrases: Hardware virtualization, Adaptable computing, FPGA architecture, Runtime reconfiguration, Operating systems

1. INTRODUCTION

The dawn of 21st Century has brought a real revolution in reconfigurable hardware. FPGAs have turned into sophisticated, flexible and extremely advanced compute fabrics which can change their own functionality on-the-fly thanks to their Dynamic Partial Reconfiguration (DPR) capability. However, the lack of standard tools and interfaces to develop reconfigurable applications limits their user base, and makes their programming difficult and unproductive. The necessity for tools and methods to exploit the advanced computation possibilities delivered by current FPGAs is now more potent than any time before. This situation is indeed similar in concept to what occurred during the infancy of electronic computers, when dealing with machine's hardware was direct and messy. Now, as then, the use of an Operating Systems (OS) is an attractive proposition to make partially reconfigurable FPGAs "programmer friendly" [Brebner 1996].

A number of research efforts have been undertaken in the last decade to build a Reconfigurable OS (ROS) for FPGAs, such as OS4RS [Mignolet et al. 2003], BORPH [So 2007], FOSFOR [Muller et al. 2005], ReconOS [Lubbers 2010] and CAP-OS [Gohringer et al. 2010b]. These are intended to provide support for dynamically swapping in and out of the FPGA a battery of computation-specific circuits ("hardware tasks") to hold a continuous stream of input operands, computation and output results. Multitasking, adaptation and specialization are key properties in ROSes, as multiple swappable

tasks can run concurrently at different positions on chip, each with custom data-paths for efficient execution of specific computations.

R3TOS is our contribution to this body of research efforts. However, in addition to the aspects listed above, R3TOS also addresses two important requirements currently highly demanded by industry: reliability and real-time, hence its name, which stands for Reliable Reconfigurable Real-Time Operating System. Notably, R3TOS deals with these issues using an approach that consists in increasing the flexibility of the system. Unlike most related work, R3TOS is aimed at exploiting most of the capabilities offered by current reconfigurable technology (e.g. slotless reconfiguration [Sedcole 2006]) with the objective of increasing performance, efficiency and reliability. For instance, hardware tasks are allocated only to functional resources, circumventing damaged parts of the chip.

While the general concepts of R3TOS have already been presented in [Iturbe et al. 2013b], this article explains its low-level implementation details. We think these are relevant for two reasons. First, the proposed implementations either solve or significantly reduce some of the currently existing problems attached to the predominantly used technology in reconfigurable computing (Xilinx FPGAs). Second, they are generic enough to be adopted in potentially any OS that uses this technology. Arguing that R3TOS is innovative in the way it exploits the reconfiguration opportunities available in Xilinx technology, we posit that most of the implementations described in this article are novel.

The remainder of this article is organized as follows. After an overview of related work in Section 2, Section 3 and 4 sum up the most important features and general architecture of R3TOS, respectively. Then, Section 5 focuses on the R3TOS hardware microkernel (HWuK), while Section 6 describes the Hardware Abstraction Layer (HAL) and Section 7 outlines the high-level Application Programming Interface (API). Section 8 gives some insights into a R3TOS prototype implementation, demonstrates its functioning in a Software Defined Radio (SDR) case-study application, and presents the obtained results with an overall performance evaluation. Finally, concluding remarks are discussed in Section 9.

2. RELATED WORK

This section covers three of the aspects that are more extensively addressed in this article: task switching, inter-task communications and synchronization. Some notions on fault diagnosis and recovery are also pointed out. Finally, the section summarizes the most important research efforts conducted to date to build a ROS on FPGAs.

2.1. Task Switching

Two main techniques have been proposed for switching hardware tasks in partially reconfigurable FPGAs. The first method consists in providing task registers and memories with extra interfaces to enable access when saving and restoring task context (e.g. [Ahmadinia et al. 2004; Kalte and Porrmann 2005; Jovanovic et al. 2007; Koch et al. 2007]). These could be implemented as memory-mapping structures or as scan chains with shadow registers [Tuan and Amano 2008]. The main advantages of this intrusive method are twofold: the fact it is largely independent of the underlying FPGA configuration architecture, and data efficiency as only the required information needs to be saved. On the other hand, its major problems are the area overhead introduced by the added interface logic, which usually reduces the maximum usable clock frequency, and the difficulty in designing standard generic interfaces for different type of tasks. The second method consists in harnessing the reconfiguration port of the FPGA for reading-back and later restoring the context information in the bitstream domain (e.g. [Simmler et al. 2000; Ahmadinia et al. 2004; Kalte and Porrmann 2005; Jozwik et al.

2010]). While transparency is the key benefit of this method, it incurs significant time overheads due to the limited access speed through the reconfiguration port and the necessity of accessing complete configuration frames, which include context information together with other state independent configuration data such as routing information. Furthermore, technical limitations also exist due to difficulties in individually accessing registers when resuming their state.

2.2. Inter-task Communications and Synchronization

The existing solutions for providing inter-connectivity to relocatable hardware tasks are based either on Network-on-Chips (NoCs) [Bobda et al. 2005; Stensgaard and Sparso 2008] or on-chip buses [Ahmadinia et al. 2005; Koch et al. 2008]. Especially interesting is the work described in [Shelburne et al. 2008], where the authors propose to emulate a NoC by harnessing the reconfiguration mechanism of the FPGA. The possibility of creating online routes among the tasks has also been proposed [Suris et al. 2008; Koch et al. 2010], but its acceptance is low due to the long time overheads when creating the routes, and difficulties when dealing with FPGA configuration, whose format is not documented. Therefore, most of currently available ROSes rely on a static communication infrastructure which interconnects all of the reconfigurable slots in the system, where the hardware tasks are allocated, and the main processor, where the software part of the OS runs, e.g. [Walder 2005; Lubbers 2010; Gohringer et al. 2010a]. With the dual objective of communicating between software and hardware tasks and abstracting hardware management, most ROSes resort to wrap the hardware tasks with software wrappers and then use existing mechanisms in standard software OS, e.g. Linux [Williams et al. 2005; Bergmann et al. 2006] or eCOS [Lubbers 2010]. As with inter-task communication, most of the existing solutions for synchronizing hardware tasks rely on static signals, e.g. [Lubbers 2010].

However, the existence of static wires crossing the FPGA surface involves important limitations in terms of relocatability of the tasks. Indeed, fully relocatable hardware tasks with support for inter-communications and synchronization currently belong only to theory. We note that this is one of the major divergences between research efforts that study the reconfigurable computing paradigm from an algorithmic perspective, usually targeting slotless reconfiguration, and the efforts that are aimed at building real prototype solutions, which only target slotted reconfigurable systems. A secondary problem is that the static communication and synchronization infrastructure occupies a significant amount of on-chip resources and usually leads to low usable clock frequencies. For instance, the NoC reported in [Bobda et al. 2005] consumes nearly all of the resources on the FPGA chip, and the bus clock frequency reported in [Sedcole et al. 2007] is only 50 MHz due to long routing delays.

2.3. Fault-Tolerance

Fault diagnosis and recovery in FPGAs is widely based on scrubbing [Heiner et al. 2009] and redundancy [Sterpone and Violante 2005; Iturbe et al. 2009]. Scrubbing consists in rewriting the configuration memory of the FPGA to correct configuration upsets. This can be done either periodically (e.g. blind scrubbing) or upon demand when an upset has been detected in any of the configuration frames. In order to detect configuration upsets, Xilinx FPGAs include a built-in logic (i.e. Frame_ECC) that is coupled to their configuration memory [Charmichael and Tseng 2009]. However, there exists a coupling problem in Virtex-4 devices, namely the Frame_ECC works one clock cycle ahead of the configuration memory. In order to deal with this problem, Xilinx have released the SEU controller macro [Chapman 2010]. Faults can also be detected and diagnosed by using Built-In-Self-Tests (BISTs). These are specifically designed

circuits to test the correct functioning of the logic and routing resources [Abramovici et al. 2004; Smith et al. 2006; Amouri and Tahoori 2011].

Finally, despite the fact that task isolation is a mandatory feature in dependable systems [Kopetz 2011], we have not found any dynamic partial reconfiguration solution that addresses this important requirement. Most of existing commercial and academic solutions do provide support for spatially isolating hardware modules within the FPGA surface (e.g. Xilinx Isolation Verification Tool [Corbett 2012], Recobus builder [Koch et al. 2008], OpenPR [Sohanghpurwala et al. 2011], GoAhead [Beckhoff et al. 2012]), but there are no solutions for ensuring isolation in the configuration domain; i.e. to ensure that the reconfiguration engine does not corrupt the system by writing configuration data in a wrong location.

2.4. Reconfigurable Operating Systems

The term ROS was coined by Dr Gordon Brebner [Brebner 1996] and it essentially refers to a software OS augmented with functions to manage reconfigurable hardware and execute hardware applications on it. The rationale of the ROS is to hide complexity by offering a set of useful services to the application developer (e.g. task switching, inter-task communication and synchronization). These services should be accessible through an API and should provide runtime support for both task management and FPGA resource management.

OS4RS was a very early ROS prototype developed by IMEC with the main focus of giving runtime support for multimedia applications [Mignolet et al. 2003]. Unfortunately, very little information is provided about its implementation and functioning. Most of the information is related to the major innovation proposed: the possibility to interrupt a hardware task and restart it in software, or vice versa. Notably, this idea has inspired later work (e.g. [Zhou et al. 2005; Pellizzoni and Caccamo 2006]).

In [Blodget et al. 2003], Xilinx released the XPART API, which was intended to ease the management of FPGA resources. Unfortunately, XPART was rapidly discontinued. In [Donlin et al. 2004], the authors wrote a Linux proc file system API for such low-level manipulations and, in [Williams and Bergmann 2004], the authors created a Linux driver for the ICAP and used it in an embedded Linux distribution, namely uClinux, running on a Xilinx MicroBlaze processor. These two can be considered the first successful attempts to make reconfigurable hardware easily accessible by a software-centric programmer, who indeed could use the ICAP from a shell script. Later, the same authors completed their work with a Linux driver that allowed FIFO-based data communications with reconfigurable hardware modules [Williams et al. 2005]. A similar ICAP driver is presented in [Donato et al. 2005]. This driver has been used to develop a Linux-based ROS with capability to manage hardware tasks as standard devices located in the `/dev/` directory [Santambrogio et al. 2008]. Finally, another Linux-based ROS is reported in [Kosciuszkiewicz et al. 2007], where hardware tasks are executed on Xilinx PicoBlaze processors and communicate with software tasks using FIFO buffers.

HybridThreads (HThreads) [Andrews et al. 2005] allows programmers to run software and hardware threads simultaneously on a CPU and FPGA. Notably, scheduling, communication and synchronization services are implemented in hardware, bringing significant performance benefits. However, as the hardware threads remain allocated on the FPGA even when they are idle (i.e. reconfiguration is not used), HThreads cannot be considered a complete ROS as it fails to manage FPGA resources (i.e. FPGA resources are not shared among threads).

BORPH [So 2007] is distributed among five Virtex-II Pro FPGAs: one of them acts as master (control FPGA) and the remaining four implement some control logic (called uK) and allocate user hardware tasks. Hence, in BORPH, hardware tasks are assigned

to user FPGAs in a one-to-one fashion, leading to a very inefficient exploitation of hardware resources. Furthermore, the amount of concurrent tasks running on the system is limited by the number of user FPGAs.

ReconOS can be seen as a porting of BORPH to a single FPGA, making special emphasis on real-time performance [Lubbers 2010]. The user FPGAs of BORPH are assigned separate reconfigurable slots in the same FPGA in ReconOS. These slots are coupled with a control logic (called OSIF), which implements the same function as uK does in BORPH. Being contained in a single FPGA, the user functions are configured through ICAP, and communications are performed through an on-chip bus running at 100 MHz. In light of achieving real-time performance ReconOS offers an eCOS-based API, which is extended with specific system calls to manage hardware tasks.

FOSFOR is very similar to ReconOS. The most significant differences to be noted are the use of an RTEMS-based API and a NoC to interconnect the reconfigurable slots [Muller et al. 2005].

A recent approach that is conceptually close to ReconOS is FUSE [Ismail and Shannon 2011]. It also relies on a slotted reconfigurable system which is implemented on a Virtex-5 FPGA and provides an embedded Linux-based API with POSIX threads running on a MicroBlaze core. Two features of FUSE are especially interesting. First, shared memories are used to exchange data between the software and hardware tasks, thus reducing data communication overheads. Second, each hardware task is associated a Loadable Kernel Module (LKM) that implements miscellaneous device driver functionality, allowing to treat hardware tasks as memory-mapped I/O device peripherals.

Finally, CAP-OS is intended to handle a variety of processors and accelerators under real-time constraints, using Virtex-4 FPGAs [Gohringer et al. 2010b]. The API offered by CAP-OS is based on Message Passing Interface (MPI), which is a language-independent communications protocol used to program parallel computers. As proposed in OS4RS, the computations in CAP-OS are to be performed either in software (i.e. by any of the processors), or in hardware (i.e. as a coprocessor). However, the currently presented prototype only supports executing software tasks, which can be loaded into the processor's program memory either through a NoC or through the ICAP. In the future, the authors expect to include the capability to configure hardware tasks upon request by the processors as well as to modify the number of processors in the system. Towards this end, bitstream relocation is pointed as necessary, leading us to understand that the current CAP-OS prototype relies on a slotted architecture.

3. R3TOS IN A NUTSHELL

R3TOS is our solution to deal with some of the major challenges that are arising in reconfigurable computing, namely to develop fault-tolerant, high-performance, (soft) real-time, low-power and adaptable reconfigurable systems from high-level descriptions.

R3TOS provides systematic OS-like support to Xilinx FPGAs, easing the exploitation of some of the most advanced capabilities of this technology by inexperienced users. Indeed, by wrapping reconfigurable hardware with a real-time software microkernel (SWuK), R3TOS creates a unified hardware-software runtime execution environment, whereby a software-centric application developer can easily use the underlying hardware resources to reliably benefit from increased computing speed with lower power consumption than achievable when using a conventional processor. In this context, R3TOS can be seen in multiple ways beyond an OS: as an abstraction model; as an interface; but above all, as an enabling solution towards mainstream usage of reconfigurable hardware.

The most important capabilities of R3TOS are listed below:

— *Real-Time*: R3TOS gives the necessary support for exploiting the inherent predictability of pure hardware in light of achieving (soft) real-time performance; i.e. good Quality of Service (QoS).
— *Dependability*: R3TOS gives the necessary support for exploiting the flexibility of FPGAs to build systems able to configure their own resources with the objective of maintaining their functionality in the presence of permanent defects and spontaneous faults. However, R3TOS is unable to recover from permanent damage affecting its own engine. In order to prevent losing the reconfiguration capability, the two ICAP instances available on FPGA are used in R3TOS so that they can diagnose and fix each other in the event of either of them being affected by a correctable upset (e.g. a SEU) [Ebrahim et al. 2012].
— *High-Performance*: R3TOS gives the necessary support for exploiting the flexibility of FPGAs to load specialized circuits upon demand, each performing a specific type of computation in an efficient way.
— *High-Level Programming*: R3TOS provides the means to make the aforementioned capabilities easy-to-use without requiring any knowledge of low-level FPGA details.

3.1. General Functioning

What makes R3TOS special is its non-conventional way of exploiting chip resources: these are used indistinguishably for carrying out either computation or communication at different times. Indeed, R3TOS does not rely on any static infrastructure apart from its own core circuitry, which is constrained to a specific region of the FPGA where it is deployed. Thus, the rest of the device is kept free of obstacles (e.g. static routes), with the resources ready to be used as and whenever needed. In this context, the partial bitstream relocation technique has been enhanced to allow hardware tasks to be mapped to different chip locations on-the-fly, with the dual objective of improving computation density, i.e. keep the tasks as compact as possible, and circumvent damaged resources. Note that permanent damage may be caused by imperfections in the fabrication process or it may emerge as the silicon ages, especially when working in extreme or harsh environments.

R3TOS is able to exploit locality (i.e. functionally related logic and data) by implementing multi-rate clocking capability [Iturbe et al. 2012]. Unlike in an ordinary FPGA implementation, where data is moved among resources which may potentially be located anywhere on the device, in R3TOS data is kept within a much smaller physical area, i.e. within hardware task boundaries, resulting in shorter paths and smaller access times. R3TOS automatically takes advantage of the reduced access times as it feeds each task with its maximum allowed clock rate to obtain the highest performance from each portion of the design [Iturbe et al. 2012]. If required, the clock frequency delivered to each task can be dynamically chosen with the objective not to increase performance, but to reduce power consumption and heat dissipation. Using this method, dynamic power consumption is automatically reduced by clocking resources locally, i.e. the resources which are not clocked only consume a leakage current. Finally, note that by doing this, unnecessary wear of the resources which are not in use is also prevented.

In R3TOS, locality is combined with global communications. Input data is firstly received in one of the edges of the hardware tasks, then processed as it flows through task data-paths and, finally, the results are stored in another edge of the tasks. This promotes the concatenation of tasks, as the input edge of one task can be placed next to the output edge of the previously executed task in the pipeline [Iturbe et al. 2011b]. This allocation strategy that leads to the creation of task chains across the device, as shown in Fig. 1, is named as *Snake*. Hence, data is automatically moved as a consequence of computing in space, i.e. data moves locally inside each task, and globally from task to task. When task concatenation is not possible, two alternative communication meth-

ods are implemented. First, on-demand interconnection wires can be created using the FPGA routing resources [Iturbe et al. 2011b]. The interconnection wires are grouped together to form the so-called Data Relocating Tasks (DRTs), which remain configured only while data is transferred between communicating tasks. In Fig. 1b, note that the tasks are typically designed to fit horizontally between two BRAM columns, and DRTs are used to vertically move data between neighbor rows. Second, the FPGA's Internal Configuration Access Port (ICAP) can be harnessed to establish on-demand "virtual" channels among hardware tasks through the configuration layer, without using any physical support [Iturbe et al. 2011a]. By using any of these methods, R3TOS is always able to provide a logical interface for the relocatable hardware tasks, regardless of their physical location on the chip, enabling inter-task communications and synchronization.

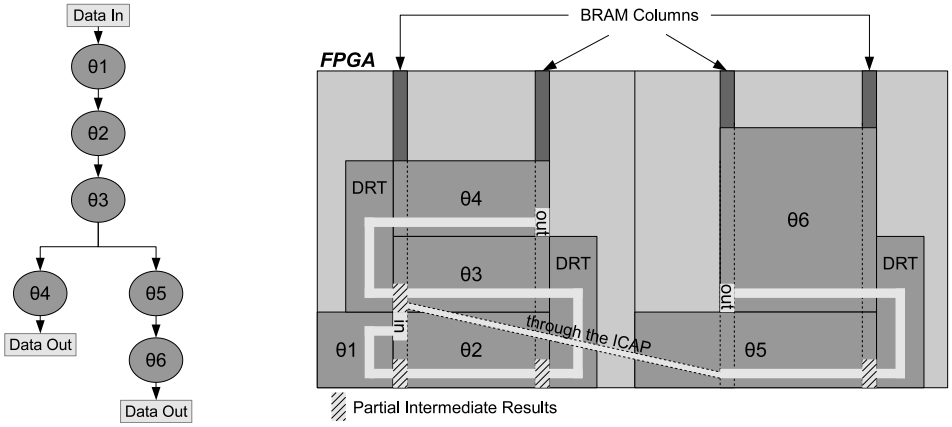


Fig. 1: *Snake*: a) Application task graph and b) Task allocation scheme

R3TOS is capable of executing redundant instances of the same (critical) hardware task in parallel at distinct positions within the FPGA (spatial redundancy) or at different times (temporal redundancy). The capability of writing identical frames to multiple configuration memory address locations provided by bitstream compression commands (i.e. MFWR commands) is exploited with the objective of speeding-up the configuration of three redundant instances of critical tasks, i.e. Triple Modular Redundancy (TMR).

Bitstream compression is also used to configure multiple identical task instances (i.e. *cloned* tasks) with very small time overheads [Ebrahim et al. 2013]. This allows to approach the building of dynamically customizable Single Instruction Multiple Data (SIMD) computers, where not only the number of (*cloned*) tasks, but also their datapath can be configured at any time to exploit both data and process level parallelism, i.e. multitasking.

The speed-up factor measured when using the task *cloning* technique ranges from 1.5x, when configuring a single *cloned* task instance, to 3.5x, when cloning up to 7 *cloned* task instances. Namely, a 2x acceleration factor has been measured when configuring three task instances, that is, when using TMR.

3.2. Flexibility vs. Performance

The limitations associated with R3TOS mainly come from the reconfiguration bottleneck provoked by the ICAP, which is shared for task (re)configuration, inter-task communications and synchronization. Nonetheless, the gained flexibility is absolutely

necessary to achieve the high fault-tolerance and adaptivity levels accomplished by R3TOS.

It is true that the performance improvement due to more flexible use of FPGA resources is constrained by the reduced communication bandwidth resulting from the lack of any pre-routed communication infrastructure in the system. But, it is also true that a communication infrastructure distributed across the entire FPGA chip and constrained within a static region usually results in large data path latencies and routing congestion, ultimately leading to low usable clock frequencies and hence, limited communication bandwidth [Sedcole et al. 2007].

Another important aspect derived from the flexibility achieved in R3TOS (i.e. slotless reconfiguration) is the possibility to exploit the entropy of the configuration information associated with hardware tasks, which results in smaller reconfiguration time in most cases. R3TOS circumvents all unnecessary configuration data added to tasks in slotted reconfigurable systems, e.g. static routes crossing the reconfigurable slots and extra configuration information resulting from the enlargement of tasks to fit the slots. This contrasts with what occurs in traditional reconfigurable systems, where configuration data is likely to be distributed among a greater number of lightly-used configuration frames.

Finally, we note there is a trade-off between inter-task communication bandwidth requirement and task granularity, which ultimately influences the reconfiguration overheads and FPGA resource requirements of the application.

4. R3TOS GENERAL ARCHITECTURE

Fig. 2 shows the general block diagram of R3TOS, which basically comprises three main parts: HWuK, main CPU and memory. The interface of the HWuK is standard enough to ensure compatibility with the most common processors and memories. External memory is necessary when developing large reconfigurable applications because of the limited availability of on-chip storage resources in current FPGAs, but it is expected that R3TOS can be implemented on a single FPGA chip in the future.

4.1. R3TOS Hardware MicroKernel (HWuK)

The HWuK includes the configuration manager and two servers which run upon it: scheduler and allocator. Each component is separately implemented to enable parallelism in the execution of the HWuK processes; i.e. while the scheduler executes the scheduling algorithm, the allocator can execute the allocation algorithm, and the configuration manager can configure a task through the ICAP. The parallel cooperation of simple components does not only result in low runtime overhead but also in acceptable area overhead; i.e. the main core of all HWuK components is a tiny Xilinx PicoBlaze processor, which requires only 96 slices in a Virtex-4 FPGA. Note that this architecture promotes the core spirit of a microkernel: upgradability (e.g. the allocator and scheduler servers can be updated to run more efficient algorithms which might be designed in the future without having to modify the rest of components in the HWuK) and scalability (e.g. multiple instances of the allocator can be used to speed-up the allocation process in very large FPGAs).

The cooperation among the HWuK components is mastered by the scheduler, with the allocator and the configuration manager acting as slaves. The communication between the components follows a very strict set of rules which are supervised by two monitors. These are capable of detecting any malfunctioning in each pair of communicating components; i.e. scheduler-allocator and scheduler-configuration manager.

The internal architecture of the HWuK components is structured around a PicoBlaze core. This executes an optimized assembly program which is based on interruptions to reduce the response time. Furthermore, each PicoBlaze uses a dedicated data BRAM

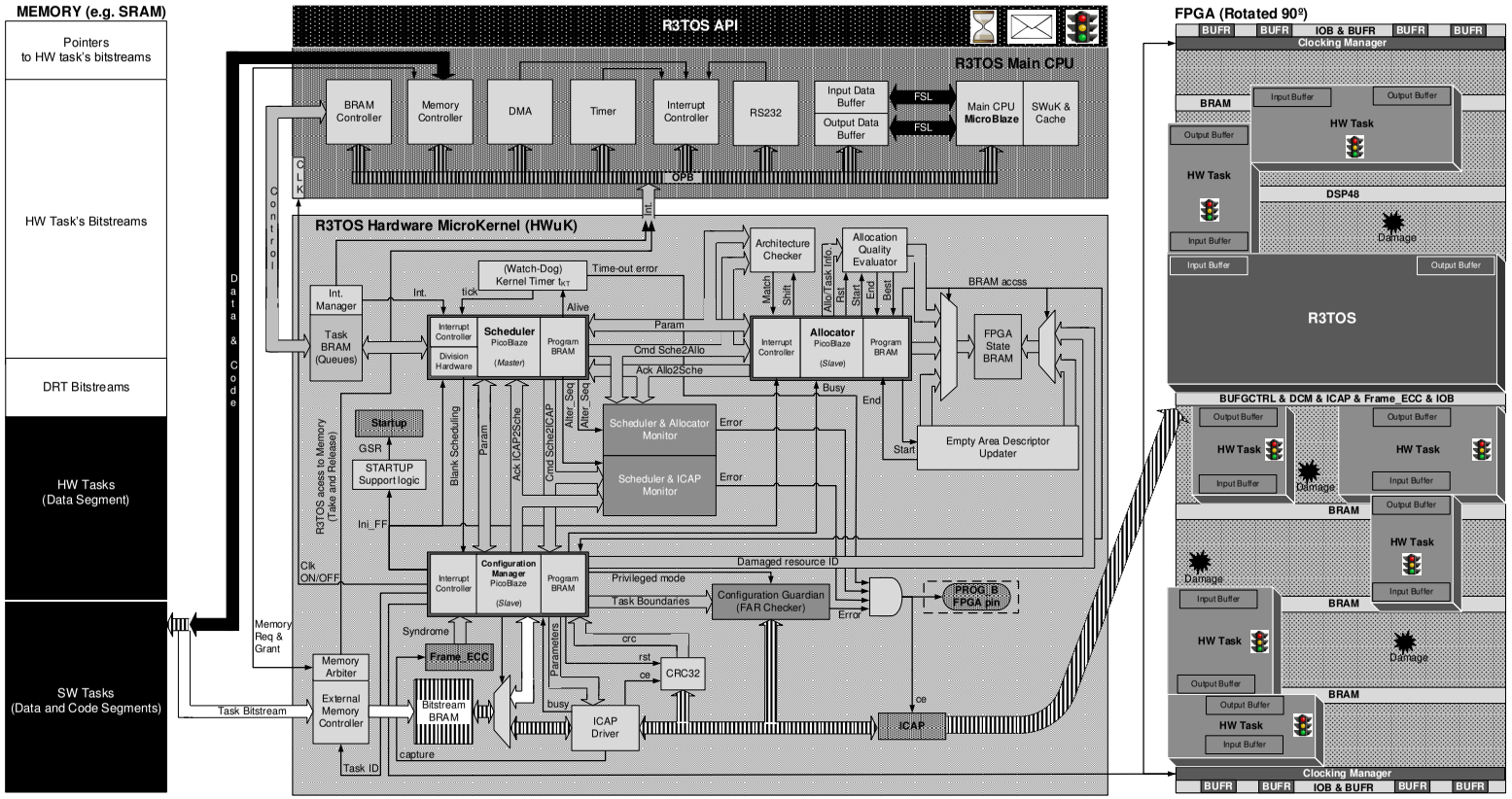


Fig. 2: R3TOS block diagram

to store the information associated with the corresponding HWuK process(es) it executes. Hence, the scheduler manages the task queues in the Task BRAM, the allocator keeps track of the available on-chip resources on the FPGA State BRAM, and the configuration manager executes pre-defined sequences of configuration commands from a Bitstream BRAM. The fact that these memories are dual-ported is conveniently exploited. For instance, the configuration manager can mark any detected damaged resource as non-usable directly in the FPGA State BRAM, and the main CPU can set in ready state any triggered task directly in the Task BRAM.

Specific to the scheduler is a timer that generates the kernel ticks. This timer is also used to supervise the correct functioning of the scheduler, which is critical as it masters the HWuK. The scheduler's PicoBlaze must generate at least one `alive` pulse within a maximum number of kernel ticks, i.e. the kernel timer acts as watch-dog timer.

Specific to the allocator are three coprocessors: (i) an architecture checker to speed-up the search for feasible allocations where the FPGA resource layout is compatible with the internal architecture of the tasks to allocate, (ii) an empty area descriptor updater to accelerate the intermediate computations required by the allocation algorithm, and (iii) an allocation quality evaluator to accelerate the making of allocation decisions.

The configuration manager interacts with the configuration-related FPGA logic. Notably, it is equipped with an finite state machine to drive the ICAP at the highest possible clock frequency, and with a CRC32 module to compute the 32-bit CRC over a set of data read from the ICAP. The finite state machine is coupled with a Configuration Guardian (CG) to ensure safe accesses to the configuration memory. Besides, the configuration manager interacts with the Frame_ECC logic to detect and localize upsets in the configuration frames. Finally, the configuration manager has access to the STARTUP primitive with the objective of initializing the hardware tasks' flip-flops with predefined values, i.e. INIT values. Since this is a very critical primitive which allows accessing internal signals to the configuration logic of the FPGA, i.e. Global Set/Reset (GSR), specific support logic is added to guarantee its safe functioning.

The HWuK also includes some extra functionality distributed across the device. This includes the Task Control Logic (TCL), which is attached to the hardware tasks, and circuitry to manage and diagnose the clocking resources, implemented next to the rightmost and leftmost IOB/BUFR columns.

4.2. R3TOS Main CPU

In the current R3TOS implementation a Xilinx on-chip processor, namely a 32-bit MicroBlaze soft-core, is used as the main CPU. This is coupled with a set of peripherals which provide additional functionality (e.g. timer or interrupt controller), connection to the external world (e.g. RS232 serial line or Ethernet), or increased performance (e.g. DMA). The peripherals are interconnected by means of an On-chip Peripheral Bus (OPB). FIFO-based high speed communications between the Input/Output Data Buffers (IDB/ODB), where data is exchanged with the tasks, and the MicroBlaze are achieved by connecting them using Fast Serial Links (FSLs). In addition, the data buffers are also accessible through the OPB bus to allow individual access to random positions. The interface with the HWuK is based on interrupts and shared memory.

The program executed by the main CPU is held in a directly accessible program memory. In the specific case of the MicroBlaze, this memory is implemented using dual-ported BRAMs and thus, it must be disabled and its clock must be stopped by the HWuK prior to accessing the content of any other BRAM within the FPGA. Otherwise, the program code executed by the MicroBlaze could get corrupted. Despite the fact this can be circumvented when using processors which do not use dual-ported BRAMs to store their program, the clock gating capability is still required when using the

STARTUP primitive. Indeed, while the GSR signal is active, the BRAMs cannot be correctly accessed and thus, any processor relying on BRAMs to store its program should be stopped to prevent executing undesired instructions. Unfortunately, clock gating by the HWuK cannot be predicted in the main CPU and therefore, may jeopardize the system's real-time performance when multiple ICAP and/or STARTUP accesses are accumulated within a short period of time. In any case, clock gating is nothing more than a work-around solution for a problem that may well be solved in newer FPGA architectures.

4.3. Memory

The external memory chip is used to store: (i) the data and code segments of software tasks, (ii) the data segments and bitstreams of hardware tasks, and (iii) the bitstreams of DRTs. Moreover, there is a pointer table located in the lowest part of the memory, which is used by the HWuK to know the exact location of each task bitstream. Aiming at achieving the highest performance when accessing the external memory, a custom memory controller capable of dealing with the pointers stored in the pointer table is included in the HWuK. In Fig. 2 note that the white parts of the memory are accessed exclusively by the HWuK while the black parts are accessed only by the main CPU. The most typical case of a single port memory is assumed in the current R3TOS implementation and hence, HWuK includes an arbiter to coordinate access from the HWuK itself and from the main CPU.

5. LOW-LEVEL HARDWARE SUPPORT FOR THE R3TOS HWUK

This section describes the most innovative circuitry developed in R3TOS, including: (i) the logic to control the execution and synchronization of the hardware tasks, (ii) the circuit to support and accelerate inter-task communications, (iii) the logic to command task preemption, and (iv) the logic to trap erroneous accesses to the ICAP which could violate the isolation of the hardware tasks in the configuration domain. Other parts are not herein described either because they do not bring any novelty (e.g. similar ICAP drivers to the one used in R3TOS can be found in [Kalte et al. 2005; Kalte and Porrmann 2006; Corbetta et al. 2009; Hansen et al. 2011]), or because they have already been reported in previous publications (e.g. scheduling and allocation engines [Hong et al. 2011; Iturbe et al. 2013a]). Finally, we propose a way to deal with the existing coupling problem of Frame_ECC logic in Xilinx Virtex-4 devices.

5.1. Task Execution and Synchronization: Task Control Logic (TCL)

Since a preliminary explanation of the TCL has already been published in [Iturbe et al. 2011a], this section highlights only the aspects that are essential to understand the R3TOS approach.

In R3TOS, the TCLs are attached to tasks' circuitry, making them self-contained and closed structures which are fully relocatable within the FPGA. They consist of: (i) a Hardware Semaphore (HWS) to enable/disable computation; (ii) an Input Data Buffer (IDB), from where data to be processed is read; and (iii) an Output Data Buffer (ODB), where the results computed by the task are stored (see Fig. 3). The results remain stored until they are required as input by a data consumer task. Hence, hardware tasks leave a *data trace* when they finish. Note that temporarily buffering the partial results along the entire chip, which is a result of the flexible allocation scheme used in R3TOS, is the natural way to exploit the distributed nature of FPGA fabrics.

The implementation of a TCL depends on the type of task it is to be attached to. For instance, large data buffers are implemented using high-density yet more scarce and location-specific FPGA storage resources (namely BRAMs), while small data buffers are implemented using low-density and more abundant storage resources (namely

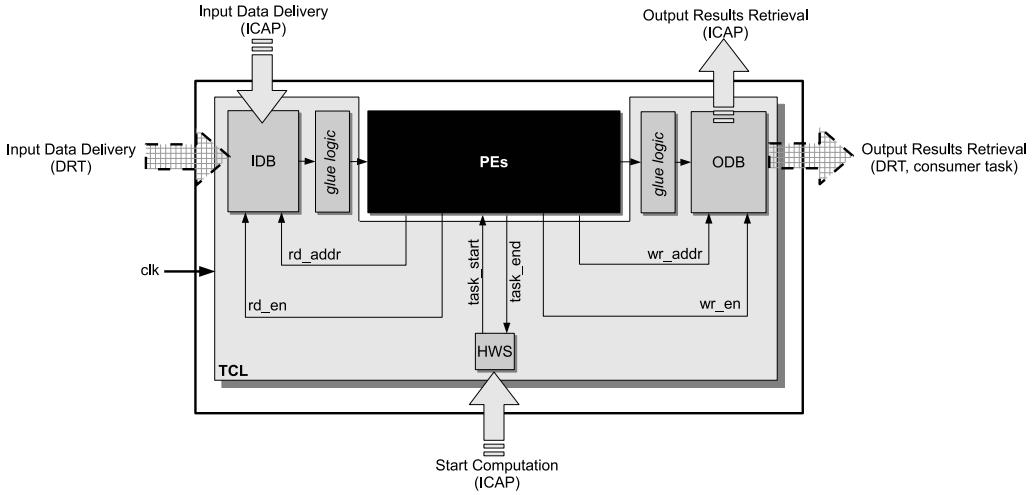


Fig. 3: Simplified generic architecture of a TCL

LUTs). Likewise, while the TCL of a data-stream processing task is suited to handle pipelined computation, the TCL of a hardware accelerator task is designed to deal with random accesses to different positions in its data buffers.

TCLs provide a means to lock physical data and control inputs/outputs of the hardware tasks to logical positions in the configuration memory of the FPGA. Since TCLs are accessible through the configuration interface whichever memory positions they are mapped to, the allocatability of the tasks is not constrained by the position of communication interfaces decided at design time anymore. Furthermore, this scheme improves multitasking capabilities as the number of tasks that can be concurrently executed on the FPGA is not limited by the amount of communication interfaces. However, we acknowledge here that the achievable data and control throughput is limited by the availability of the reconfiguration port(s) in the FPGA. In Fig. 3, the solid-line arrows represent the logical access through the ICAP to the information stored in the data buffers, and the broken-line arrows refer to the physical access through the functional layer to BRAM-based data buffers in the *Snake* approach. The latter access is performed either by DRTs or directly by data consumer tasks when reusing partial results that are stored in TCL's ODB.

The synchronization needed for coordinating access to data buffers is provided by the HWS included in the TCL. The HWS is the same for all types of tasks and it is implemented on a single LUT-RAM. This is set to '1' by HWuK through the ICAP to start computation, and it is automatically set to '0' by the task itself when the results are ready in the ODB [Iturbe et al. 2011a]. Hence, the HWS can be polled from the HWuK through the ICAP to detect a computation completion. The HWS acts as local reset for the registers of a particular hardware task while data transactions are carried out to/from TCL's data buffers. Note that a good practice for FPGA design is to use a reset input in all of the registers to initialize them with a known state at startup. The startup state of registers is coded in the SRMODE bits in the configuration bitstream. In addition, hardware description languages also allow for initializing the registered signals to specific values without using any reset. In the latter case, the specified initialization values are coded in the INIT bits in the configuration bitstream and loaded in the actual registers by issuing a GRESTORE command when the configuration bitstream is transferred to the FPGA's configuration memory. By using the STARTUP

primitive, and conveniently using the masking possibility of flip-flops, R3TOS is able to perform a local GRESTORE operation exclusively affecting the registers contained in a particular hardware task[1]. This mechanism is the basis for task preemption in R3TOS.

5.2. Inter-task Communications: Data Relocation Tasks (DRTs)

DRTs include all the necessary circuitry to move data from a source BRAM-based ODB to a target BRAM-based IDB through the functional layer of the FPGA. This includes both the logic to drive the BRAMs and the wires to connect them. DRTs are managed as if they were standard computing tasks, where their height (h_y) and width (h_x) are equal to the vertical and horizontal distance between the source and target BRAMs they connect. As standard computing tasks, they can be used only when the region where they are to be allocated is completely free. The lack of flexibility of using DRTs to communicate tasks when compared with online routing is to be compensated by providing a set of differently sized and shaped DRTs to cover several source-target BRAM interconnections, e.g. BRAMs located in the same column, two columns apart, etc. In any case, the number of DRTs is limited in order to reduce both the amount of memory necessary to store their bitstream configurations and the number of positions to be evaluated when allocating a task.

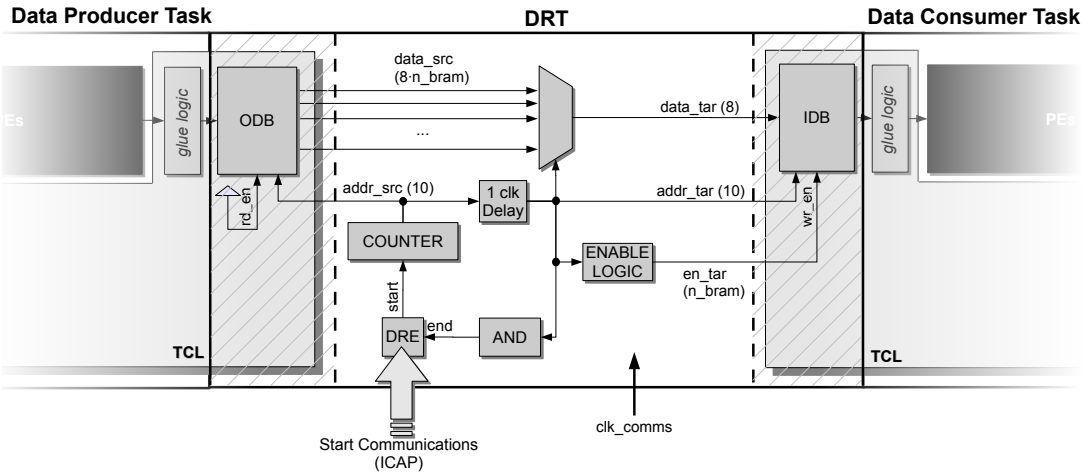
The way that a DRT is attached to data producer and consumer tasks is depicted in Fig. 4a. The stripped regions represent the routing of the BRAMs, which is dynamically switched to permit access to the memories either from the computing tasks or from the DRT. Note that BRAMs are the left and right boundaries for both DRTs and computing tasks. Indeed, DRTs and computing tasks are assigned different partially reconfigurable regions and synthesized following the Xilinx partial reconfiguration flow, thus ensuring no routing conflicts occur when switching the BRAM interconnections, i.e. the Programmable Interconnection Points (PIPs) that are active in computing tasks are not active in DRTs and vice versa. In order to prevent corruption of the data stored in memories due to unexpected early activity while switching BRAM interconnections, the clock signal delivered to the BRAM column is first stopped.

Access to target BRAMs from the consumer task is automatically gained when configuring its bitstream, i.e. BRAM interconnection frames are overwritten (see Fig. 4b). It must be ensured that all data has been transferred by the DRT by the time the consumer task is completely configured, as both processes are carried out at the same time. In order to leave the FPGA in the same state as it was before using the DRT, thus preventing future routing conflicts, two steps must be made: (i) the DRT must be deallocated by blanking all the necessary configuration frames and, (ii) the TCL of the temporarily modified data producer task must be restored by rewriting the original BRAM interconnection configuration.

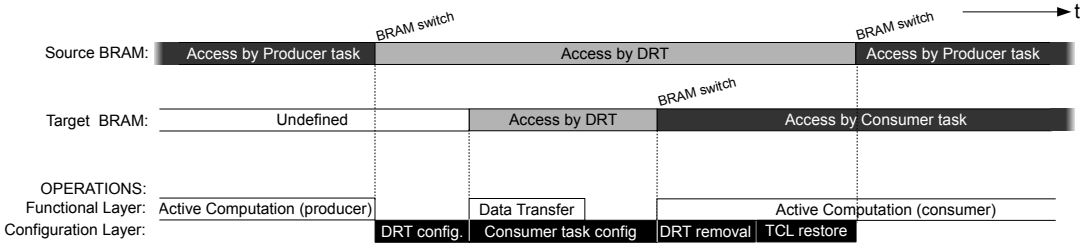
In order to control communications from the HWuK, a HWS-like LUT-RAM is included in DRTs. This is named as Data Relocation Enable (DRE) and it is activated through the ICAP only when the source and target BRAMs are correctly connected by means of the DRT. Then, data is sequentially read from the producer task's ODB (source BRAM) and copied, one clock cycle later, to the consumer task's IDB (target BRAM). This process is repeated until the last data is copied. Afterwards, the DRE is automatically disabled by the DRT's own logic.

As shown in Fig. 5, the logic to drive the BRAMs is mapped to a single CLB column, namely to the CLB column located next to the source BRAMs. The rest of the

[1]Xilinx has made available a specific constraint from Virtex-6 onwards, RESET_AFTER_RECONFIG, to automatically manage the flip-flop initialization masking bits in a partial bitstream so that only the registers included within the reconfigurable area are initialized after reconfiguration



(a) DRT logic (the number between parentheses is the bus width)



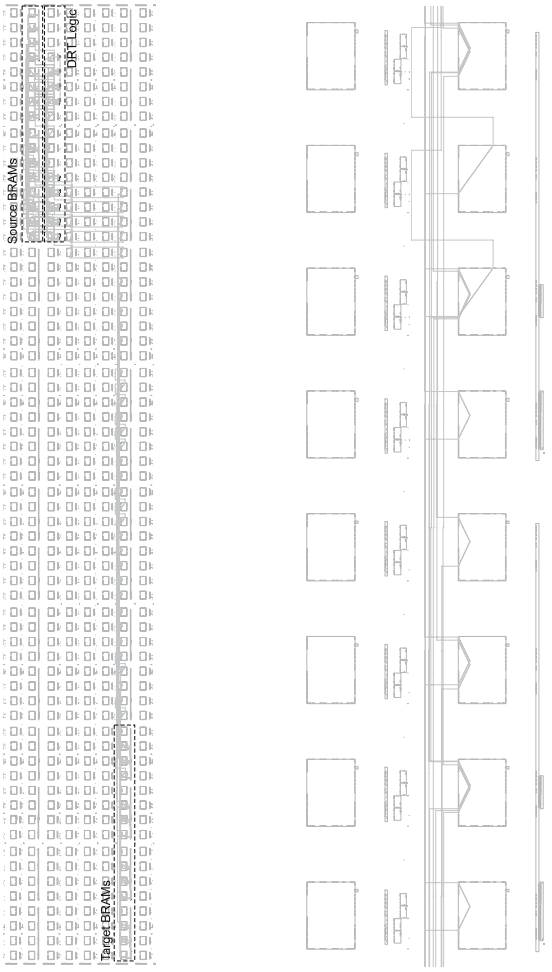
(b) Management of DRTs: Timing diagram

Fig. 4: DRTs

DRT is free of logic, it only includes the wires to connect the source and target BRAMs together. In order to reduce the amount of wires, and thus reduce configuration time, the width of the BRAM port is set to 8-bit when using DRTs. Furthermore, an input multiplexer is used to select the data delivered by only one of the source BRAMs at any time, i.e. the DRT receives $8 \cdot$ `n_bram` input data wires, which are time-multiplexed using only 8 wires. Hence, the total amount of wires to be routed to the target BRAMs is: 19+`n_bram`, where `n_bram` are the BRAM enables, 11 are address and 8 are data. Using this scheme and working at 100 MHz, it is possible to transfer the content of up to 4 BRAMs within less than a hundred of microseconds. In any case, higher amounts of data can be transferred within the same time by increasing the frequency of `clk_comms`, which indeed can be done as for the clock delivered to standard computing tasks.

The benefit of the above implementation is that the configuration information of the DRT is mainly concentrated in the 20 interconnection frames associated to the source and target BRAMs and the 22 frames of the CLB column where the logic is implemented. The remaining configuration information of the DRTs consists of a set of active PIPs, which can be grouped into a reduced number of frames by exploiting the regularity of the FPGA routing structure (see Fig. 5) with the ultimate objective of reducing the amount of time needed to configure the DRTs.

Fig. 6 shows the requirements imposed by DRTs in terms of occupied slices and amount of frames necessary to be configured. Note that only DRTs that connect BRAMs in the vertical direction are shown in this figure (i.e. $h_x = 1$) as this is the



(a) (b)

Fig. 5: DRT implementation: a) Bird-eye view and b) Detailed view

general use of DRTs (see Fig. 1). Similar results are expected for DRTs that connect BRAMs in the horizontal direction as the FPGA routing pattern (e.g. hex and long lines) is the same in both vertical and horizontal directions. Also note that the number of active PIPs do not significantly increase as the (vertical) distance between the BRAMs to connect increases, i.e. as h_y increases. This is the result of three factors: (i) use only 8 multiplexed data wires, (ii) exploit the regularity of FPGA's routing structure, and (iii) use hex lines when the BRAMs to connect are located in different rows. On the other hand, the number of active PIPs and frames inevitably increases when using a larger amount of data buffers (i.e. more BRAMs) as the DRT receives more input data signals and also uses more slices to implement the input multiplexer and the rest of its internal logic. Based on this figure, DRTs make it possible to accelerate

data transfers among BRAMs by an approximate factor of[2]: $\frac{128}{85}$ (1.5x when n_bram=4), $\frac{256}{112}$ (2.3x when n_bram=8) and $\frac{512}{179}$ (2.8x when n_bram=16).

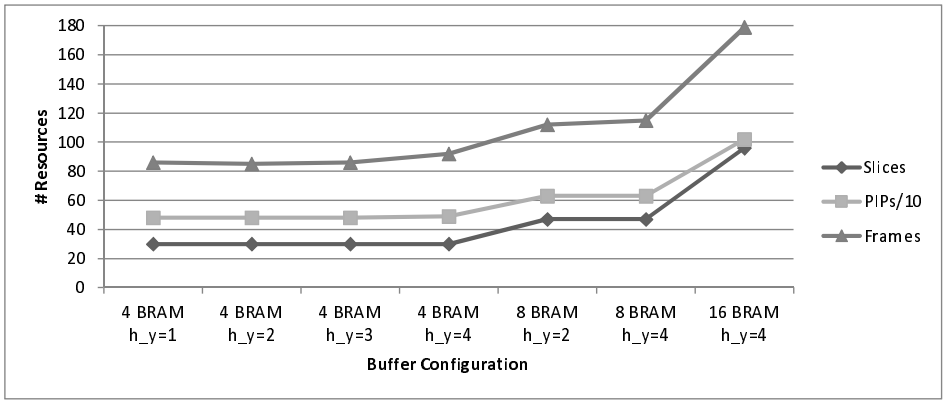


Fig. 6: Resource and configuration requirement for different DRTs ($h_x = 1$)

5.3. Task Preemption: STARTUP Support Logic

This section describes the logic that has been added to R3TOS in order to safely use the STARTUP primitive; i.e. to avoid corrupting any part of the system when accessing the GSR signal. Namely, we have noticed that BRAMs give all zeros while the GSR signal is active. A possible explanation for this might reside in the fact that BRAMs are coupled with registers at their output. As a result, the STARTUP primitive cannot be directly managed by one of the HWuK PicoBlazes, which relies on BRAMs to execute its program, but by a specific logic. This logic basically generates a 2 clock cycles GSR duration pulse upon request (ini_FF signal) from the configuration manager's PicoBlaze (see Fig. 2). Moreover, the latter PicoBlaze is also responsible for appropriately configuring the flip-flop masking bits in order to restrict the initialization operation to the flip-flops contained within the specific FPGA region.

It must be noted that all zeros read from BRAMs while enabling the GSR signal are decoded as `LOAD s0, 0` instruction by the PicoBlaze and hence, the value stored in the s0 register must be saved and restored immediately before and after triggering the GSR signal. As shown in Fig. 7, the code to perform these operations gives rise to a specific Interrupt Service Routine (GSR_ISR) in the R3TOS PicoBlazes. Therefore, the ini_FF signal is used as an interrupt in all of the PicoBlazes in the HWuK, and the delay introduced by the STARTUP support logic when generating the GSR pulse allows enough time for the PicoBlazes to enter the GSR_ISR.

Besides, the zeros read from the BRAMs while enabling the GSR signal can lead to undesired instruction execution in the main CPU (e.g. MicroBlaze), which might be harmful to the system. In order to secure the main CPU, the configuration manager's PicoBlaze keeps its clock stopped while performing the GSR operation. Likewise, the

[2]The numerator is the sum of the frames that need to be read from the producer task's ODB and written to the consumer task's IDB when DRTs are not used. Note that each BRAM column (4 BRAMs) has 64 data frames. The denominator is the number of frames that need to be processed when using DRTs, which is shown in Fig. 6.

```
-- Conf. Man. PicoBlaze --
; Disable CPU's clock
; Disable HW tasks' clock
; Enable Initialise FFs
LOAD   s0, s0
LOAD   s0, s0
; Disable Initialise FFs
; Enable CPU's clock
; Enable HW tasks' clock
-------------------------

GSR_ISR:
   STORE  s0, S0_stack
   LOAD   s0, s0
   LOAD   s0, s0
   FETCH  s0, S0_stack
   RETURNI ENABLE

ADDRESS 3FF
   ; Check Initialise FFs
   JUMP GSR_ISR
```

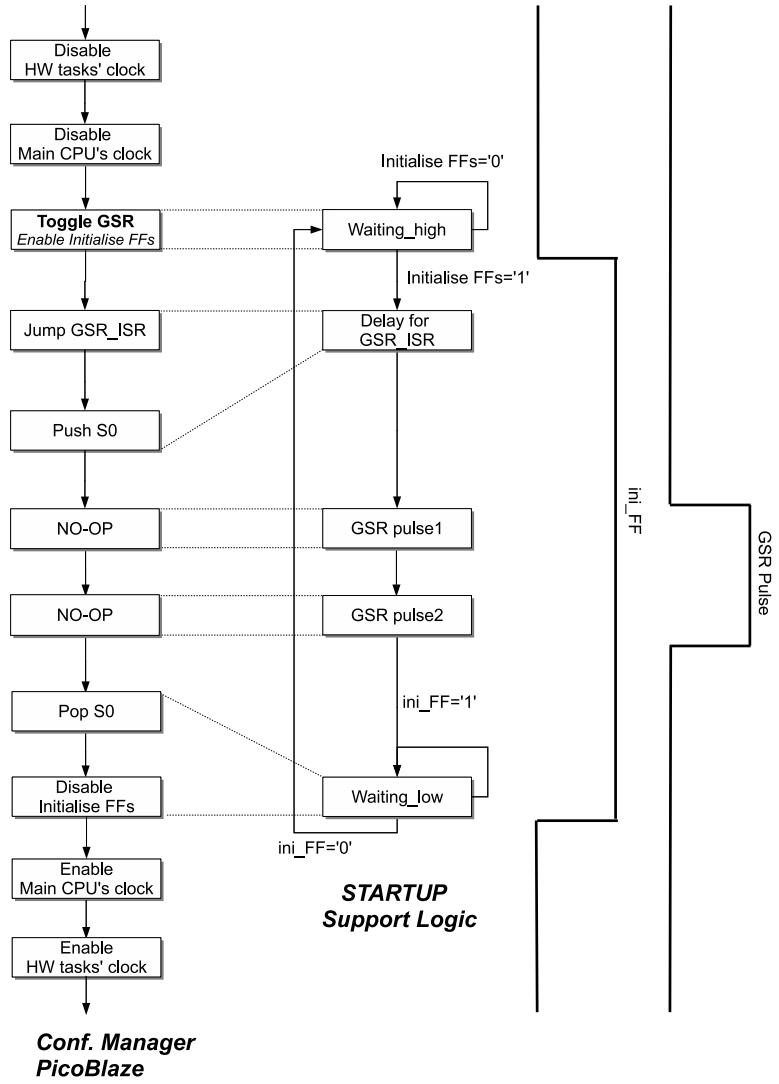


Fig. 7: STARTUP support logic

clock delivered to the hardware tasks that use BRAMs is also stopped prior to activating the GSR signal. Both the clock delivered to the main CPU and to the tasks are again resumed by the configuration manager's PicoBlaze when returning from `GSR_ISR`.

5.4. Task Isolation in the Configuration Domain: Configuration Guardian (CG)

The objective of the CG is to prevent the corruption of the system due to erroneous accesses to the ICAP. Besides, the CG increases the error detection coverage as the erroneous ICAP accesses are typically the result of a wrong configuration state of R3TOS itself. The CG prohibits access to the resources assigned to HWuK except when the *privileged mode* is enabled, e.g. to scrub a dormant upset affecting the HWuK. Therefore, the CG conceptually acts as the Memory Protection Unit (MPU) in a conventional

processor, playing a vital role to isolate the hardware tasks in the configuration domain.

The CG circuit stores in four (only readable) registers the coordinates of the upper-left and bottom-right vertices of the region where the R3TOS circuitry is implemented on the FPGA. The latter registers are initialized with the appropriate values at design time. Since access to any frame within this region is only allowed in privileged mode, any attempt to access them without having activated the privileged mode first is considered erroneous. Likewise, when in privileged mode, any access to a frame out of the region assigned to R3TOS is considered erroneous. Besides, the CG circuit includes four writable registers to load the coordinates of the upper-left and bottom-right vertices of the FPGA region assigned to the task to be accessed. The CG snoops the data transmitted through the ICAP data bus to detect any writing operation to the Frame Address Register (i.e. 0x30002001 value), and to capture the subsequent frame addresses. If the ICAP access is to a frame out of the specified region in the coordinate registers, then that access is considered erroneous, and a full FPGA reconfiguration is triggered via its PROG_B pin to fix the logic that has provoked the error. In the unlikely case that the 0x30002001 value transmitted to the ICAP is a raw configuration data and does not correspond to a FAR writing operation, the CG may produce a false-positive error detection. To avoid false-positive error detections, the CG should be able to process the header of the configuration data packets, which would make it more complex and thus more prone to error, i.e. to produce false-negatives. We think it is preferable to use a simpler CG as it currently fits in only 38 slices.

5.5. Fault Diagnosis

In order to overcome the existing coupling problem in the Frame_ECC logic of Virtex-4 FPGAs, the ICAP driver captures the ECC syndrome given by the Frame_ECC logic when reading the 40th frame word through the ICAP. Besides, the ICAP and Frame_ECC logic are synchronized prior to checking the ECC code of any frame. As shown in Fig. 8, $41 - N_{Rd}$ words are read to re-synchronize both parts, where N_{Rd} is the amount of read operations performed since the last synchronization and is reset to 0 every time it reaches 41. We note that our solution permits to avoid the use of the proprietary Xilinx SEU controller macro, allowing for the development of a single centralized controller that could implement all of the reconfiguration-related functionality. In the specific case of R3TOS, this results in a simpler and less error prone configuration manager implementation.

6. R3TOS HARDWARE ABSTRACTION LAYER (HAL)

The HAL turns the vast and complex FPGA hardware into an easy-to-use computing resource which can operate in connection with any CPU / software OS required by the developer. It is mainly based on the basic functionality implemented by the configuration manager, which is extended to build a set of more capable services. Namely, the configuration manager implements up to six reconfiguration related functions: (i) Partial Bitstream Relocation (PBR), (ii) Read-back Frames (RBF), (iii) Write Multiple Frames to a Single location (WMF2S), (iv) Write Single Frames to Multiple locations (WSF2M), (v) Write Single Frames to a Single Location (WSF2S) and (vi) Blank Frames (BlF). The configuration command sequences to be executed for each function are stored as *configuration templates* in the Bitstream BRAM. These templates are adjusted by the configuration manager's PicoBlaze with the specific parameters of the operation that needs to be performed at any time prior to sending them to the ICAP; i.e. some of the positions in the Bitstream memory are editable while the others store pre-defined configuration commands.

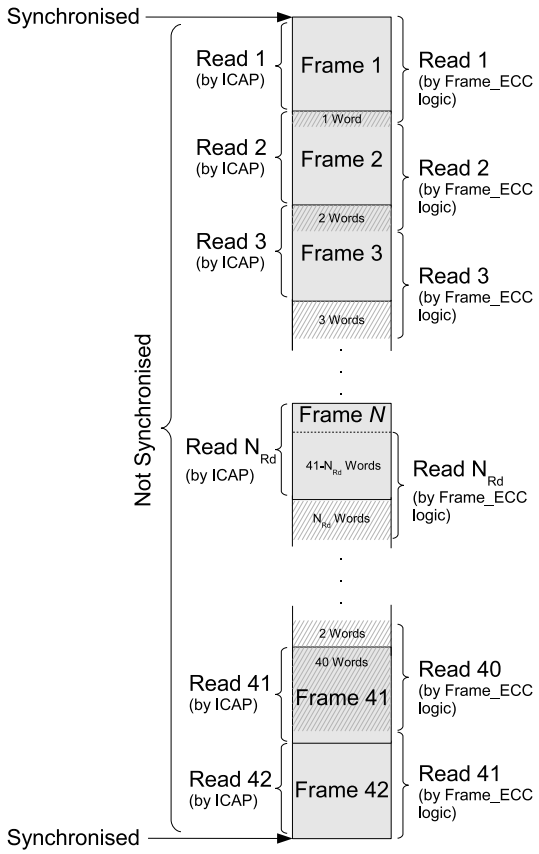


Fig. 8: Synchronizing Frame_ECC logic and ICAP in Virtex-4

Note that there are up to three different writing functions, i.e. WMF2S, WSF2M and WSF2S, in order to take advantage of bitstream compression (i.e. MFWR commands) in different situations. Compression is used when writing single frames (e.g. enable a HWS or scrub an upset), or when when writing the same set of frames to multiple positions on the FPGA (e.g. when copying the BRAM content frames to three redundant task instances). In the former case, compression is beneficial as there is no need to add any extra pad information to the frame data, while in the latter case the benefit comes from the possibility of writing the same frame to three locations at the same time. However, the overhead due to the fact that MFWR commands apply in a frame-by-frame fashion makes compression inefficient when writing a large set of frames to a single position on the FPGA (e.g. when copying the BRAM content frames to a single instance of a task). Additionally, the BlF function also uses bitstream compression to write all-zeros in the configuration frames to be blanked. Note that this is possible to be done in R3TOS as there are no static signals that must be preserved when blanking any FPGA region.

An itemization of the functions which are triggered in the configuration manager when executing each HAL service is shown in Table I.

Service	PBR	RBF	WMF2S	WSF2M	WSF2S	BlF
Task (Re)Allocation	✓	✓		✓	✓	
Task Deallocation						✓
Task Preemption	✓	✓		✓	✓	
Inter-task communication	✓	✓	✓	✓	✓	
Clock Management		✓		✓	✓	
Fault-Diagnosis		✓		✓	✓	✓

Table I: HAL services mapping to functions executed in the configuration manager

6.1. Task (Re)Allocation Service

The task allocation service allows for configuring a single instance of a task, a triple instance of a critical task, or a set of *cloned* tasks, with `TaskID=`i in a target location (X,Y) and running at a specific $f_{CLK,i}$ clock frequency. Once the partial bitstream of a task is successfully uploaded to the FPGA's configuration memory, other HAL services are invoked to deallocate overlapping tasks (i.e. task deallocation service, see section 6.2), to deliver input data to the allocated task (i.e. inter-task communication service, see section 6.3), and to set-up the appropriate clock frequency for the task (i.e. clocking management service, see section 6.4). The HWS of the task remains disabled while all these services are executed, i.e. the task is in reset state. Afterwards, the GSR is toggled to force all registered signals in the task to be loaded with their INIT values, which may be either the original initialization values or the context values previously saved. Finally, the HWS is enabled and the task starts performing active computation.

6.2. Task Deallocation Service

Task deallocation is necessary to avoid routing conflicts when a task θ_i is allocated on a location that overlaps with the footprint of a previously executed task θ_j. Note that in this situation, some of the resources assigned to θ_i are likely to be connected to other resources located out of its boundaries, but previously contained within the boundaries of θ_j. These undesired connections might interfere with the normal operation of θ_i, being a potential threat for the correctness of computation and might even damage the FPGA by setting short circuit situations. In order to remove the latter connections, the configuration information of θ_j that is not overwritten when loading the bitstream of θ_i must be blanked prior to enabling θ_i's HWS. The blanking operation, however, does not affect the BRAM content frames or LUT content frames as the data traces must still remain on FPGA after deallocating the producer tasks. These are blanked when all consumer tasks have accessed them.

The task deallocation service is mainly based on the frame blanking function (BlF) implemented by the configuration manager. While this function writes all-zeros to all of the (minor) frames included in a specific FPGA column, the task deallocation service gives the necessary support to blank a given rectangular region within the FPGA surface. Hence, the BlF function is repeatedly executed with the appropriate frame address and amount of minor frames in each consecutive resource column within the region to be blanked. Since the latter information is obtained from an FPGA descriptor held in the lowest part of the Bitstream BRAM, the only input parameters to this service are the coordinates of the two corners of the region to be blanked, i.e. upper-left and bottom-right corners.

6.3. Inter-task Communication Service

In general, the inter-task communication service is aimed at providing support for reallocating the content of the ODB of a producer task θ_i to the IDB of a consumer task θ_j. So far we have seen that this operation can be conducted in various ways in

R3TOS, requiring different functions to be triggered in the configuration manager in each case.

The most generic communication method consists in reading-back the frames associated to the ODB of the data producer task, and writing them to the IDB of the data consumer task(s). This can be done either using the WMF2S function, when dealing with non-critical consumer tasks, or using the WSF2M function, when dealing with triplicated critical consumer tasks or *cloned* task instances. Unlike when saving the context of a task for an undetermined amount of time, when carrying out inter-task communications, the exchanged data is read-back and immediately written to the target location, using the Bitstream BRAM to temporarily buffer it.

The inter-task communication service envisages the possibility of exchanging data among buffers implemented using different FPGA resources. This could be useful when communicating tasks which have not been developed according to the R3TOS guidelines (e.g. using flip-flops to store data) with tasks that do follow the R3TOS recommendations, and thus use exclusively LUTs and/or BRAMS.

In order to adapt the format of the data to the needs of different FPGA resources, the read-back frames need to be processed in the Bitstream BRAM, resulting in longer communication overheads. While LUTs and flip-flops store their content "in clear", i.e. the 16 data bits stored in a LUT occupy consecutive positions within the bitstream, BRAM content data is distributed along a set of frames in a non-trivial way. We only know that 256 consecutive data bits and 32 consecutive parity bits are mapped to the same BRAM content frame. In light of this, exchanging data between LUTs and flip-flops is trivial: 16 flip-flops are grouped together to form the content of a LUT and, conversely, each of the 16 bits of a LUT are copied to the INIT bits corresponding to separate flip-flops. However, exchanging data between LUTs or flip-flops and BRAMs is more complicated, requiring to deal with the "obscured" data format used to store the BRAM content in the bitstream. The easiest and fastest way to deal with this issue is to use a coding-decoding BRAM, that can be the Bitstream BRAM itself. Hence, flip-flop or LUT data is extracted from the original location within the read-back frames and written-back "in clear" to 256-bit length segments within the Bitstream BRAM. Afterwards, the memory segments are read-back through the ICAP and finally written to the target BRAMs, i.e. to the IDB of the data consumer task. The latter access through the ICAP is necessary to convert the data "in clear" to the suitable format demanded by the BRAMs. In order to exchange data in the opposite way, content frames are read-back from the source BRAM, i.e. data producer's ODB, and written again to frames which correspond with different memory segments within the Bitstream BRAM. Note that it is then possible to access data "in clear" through the latter BRAM's ports. Finally data is rearranged to the suitable location within the frames prior to being copied to the LUTs or flip-flops.

The remaining two communication methods are mainly based on the partial bitstream relocation function (PBR) implemented by the ICAP driver. The first method consists in allocating the data consumer task in such a way that it has direct access to the BRAMs where input data is stored. The second method requires the allocation of a DRT prior to configuring the data consumer task. The inter-task communication service automatically manages the DRTs, selecting the appropriate one for each communication situation, retrieving its associated partial bitstream from the external memory, and once it is uploaded to the FPGA's configuration memory, invoking the clocking management and the task deallocation services as needed.

6.4. Clocking Management Service

In R3TOS each task receives as many clock signals as clock regions it spans in the FPGA. These clock signals are synchronized because they are derived from the same

(global) clock signal (i.e. BUFG). While all of the clock signals must be of the same frequency, they can be routed through different regional clock nets in each clock region. This depends on the occupancy of the FPGA as well as on any existing damage in the clock-tree, and requires the capability to route the clock signals inside the tasks at runtime. Indeed, R3TOS implements a complete online clock routing mechanism, where some parts are autonomously managed based on reliability premises (e.g. BUFGCTRLs), and the others are managed by either the clocking management service (e.g. BUFRs) or by the task allocation service (e.g. clock routing inside the tasks). This approach contrasts with the usually chosen option of using a single clock source per task. Indeed, the vast majority of reported reconfigurable systems do not use regional clocking resources, and most of the systems which do use them are not capable of routing the regional clock signals inside the tasks (e.g. [Jara-Berrocal and Gordon-Ross 2010]). The latter approaches include BUFRs as a component of the tasks and therefore, the clock routing remains unchangeable inside them.

While the clocking adjustments are sequentially performed, i.e. only one frame can be read or written through the ICAP at a time, the HWS allows to enable all of the clock signals which feed the same task at a time, circumventing any potential synchronization problem. For simplicity and predictability, the clock signals are not modified during the execution phase of the tasks, neither their rate nor their routing.

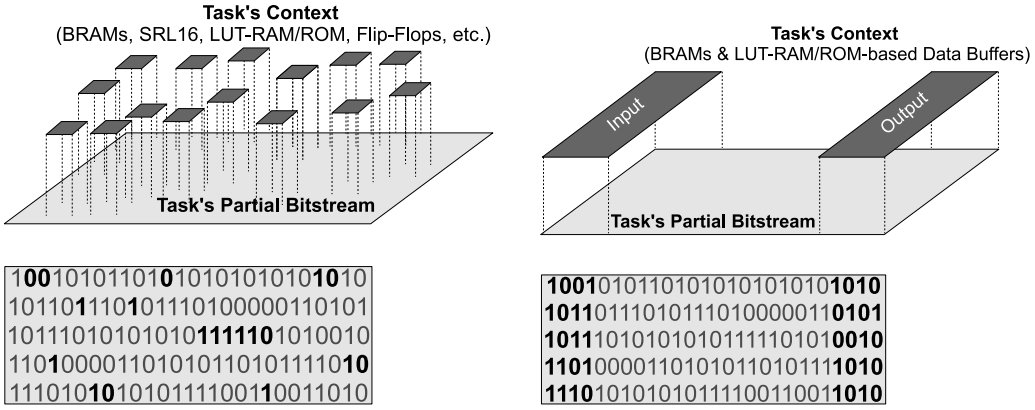
When only one frequency is needed in a clock region, the other BUFR is disabled to reduce the power consumption. Likewise, when the two BUFRs in a clock region are damaged, a spare clock signal from any of the 4 BUFRs located in the top or bottom neighbour clock regions is switched. This is done by appropriately configuring the PIPs that select the input connection of the regional clock lines [Iturbe et al. 2012]. This novel use of the branched clock-tree available in modern Xilinx FPGAs significantly increases the reliability of the system.

6.5. Task Preemption Service: Context Save and Restore

Preemption is not currently allowed in the execution phase of the tasks. Instead, the duration of execution is limited and usually known, i.e. the amount of time needed by pure hardware to come up with the results can usually be predicted with precision. As a result there are a set of pre-defined instants when the tasks finish a partial computation and the results computed by them can be accessed. Note that when the tasks are *reentrant*, accessing the partial results is conceptually similar to a context save as the execution of these tasks can be later resumed by delivering the partial results again to them, i.e. by restoring the task context. Unlike in a spontaneous task preemption, where the state of all memory elements and registers included in the task must be saved (see Fig. 9a), the amount of results to be saved in R3TOS is more reduced and localized in specific configuration frames, limited to the data stored in the task's data buffers (see Fig. 9b).

Data buffers are implemented using either LUT-RAM/ROMs or BRAMs in R3TOS in order to enable access to them directly in the FPGA's configuration memory. However, when the implementation of these buffers is not up to the application developer (e.g. the tasks are developed by a third party company or a high-level design tool which does not allow to specify the implementation details is used), it might happen that part of the data to be exchanged is mapped to flip-flops or to disjoint, or even unknown, resources within the tasks. In the latter case, there is a need to save the state of all resources which could potentially store user information, including flip-flops, BRAMs, SRL16s and LUT-RAM/ROMs.

The operation sequence to save the context of a task is as follows. First, the task is stopped. This can be done by disabling the BUFR if the task spans a single row. Otherwise, a HWS-like mechanism must be used to allow glitchless task stopping, namely an



(a) Traditional task context

(b) Task context in R3TOS

Fig. 9: Task context

ICAP writable LUT-RAM connected to the clock enable inputs of all task's sequential components. Then, a GCAPTURE command is issued to make the content of flip-flops accessible in the FPGA's configuration memory. Finally, task's configuration information is read-back. This information includes both exclusively configuration data (i.e. the task's partial bitstream) and user data (i.e. the task's context). Since the read-back information is likely to be large, it should be temporarily stored in the external memory. In some cases, e.g. when dealing with reentrant tasks, this information could overwrite the original task's partial bitstream stored in the latter memory. If this is the case, the task allocation service presented in section 6.1 can be used to restore the task context. Indeed, the state of BRAMs, SRL16s and LUT-RAM/ROMs is automatically restored when loading the saved context without requiring any other action to be taken, and the state of flip-flops is restored as a result of triggering the GSR signal. Thus, the task preemption service enables writing only for a set of flip-flops contained within a particular FPGA region where a task is to be allocated prior to restoring its context, and disables it when finishing, prior to enabling the HWS of the task.

6.6. Fault-Diagnosis Service

R3TOS relies on the Frame_ECC codes included in the configuration frames to periodically detect kernel configuration errors and to correct single upsets (e.g. SEUs). After correcting an upset, the frame is read-back again to check whether the error continues to exist. If so, permanent damage is assumed and the system initiates a fail-safe shutdown.

Computation errors are detected when any of the three redundant instances of a critical task computes a different set of results (i.e. value domain errors), or when the HWS of a task that should have finished its computation does not indicate so (i.e. time domain errors). R3TOS diagnoses the source of computation errors and, in case they are due to permanent damage on the chip, it prevents the future use of the damaged resources.

Damage in the clocking infrastructure is detected by checking all of the diagnostic circuits associated to the BUFRs that deliver a clock signal to the erroneous task instance [Iturbe et al. 2012]. If this is not the cause of the error, R3TOS proceeds to carry out an exhaustive test of all routing wires and logic resources included in the FPGA region where the erroneous task instance was allocated, i.e. Region Under Test (RUT). In order to do so, R3TOS relies on using Built-In-Self-Test (BIST) circuits. We note

that any BIST circuit, such as those described in [Abramovici et al. 2004; Smith et al. 2006; Dutt et al. 2008; Amouri and Tahoori 2011], is potentially amenable to be used in R3TOS. The only requirement imposed by R3TOS is the necessity to command the BIST circuit by means of a HWS and access the diagnosis results remotely through the ICAP. Since the BIST circuits remain configured in the FPGA only for a limited amount of time, i.e. while carrying out the diagnostic test, they are named as Resource Diagnosing Tasks (RDTs). These are managed as standard tasks, that is, they are allocated and provided with a clock signal using the previously explained HAL services. With the dual objective of speeding-up the diagnostic test execution and covering all potential different sizes and shapes of RUTs, R3TOS relies on a single basic BIST circuit that is replicated along the RUT using the task *clonaing* capability implemented by R3TOS. The BIST circuit instances tiled in the RUT are then simultaneously activated in order to perform concurrently.

7. R3TOS APPLICATION PROGRAMMING INTERFACE (API)

R3TOS is completed with a POSIX-like API, which provides high-level software-centric users with a familiar way to access the low-level services implemented by the HWuK, i.e. HAL, and ultimately to exploit the FPGA resources. The HAL is thus wrapped with a software OS layer, the SWuK, which is executed on the main CPU. On the whole, the combination of R3TOS HWuK and SWuK results in a good framework to develop hardware-software hybrid applications. However, the achieved higher abstraction level is traded-off with a loss in performance and an increase in power consumption, as the CPU needs to execute extra OS processes. Besides, the system becomes more complex, and thus more error prone.

7.1. The R3TOS Main CPU

We will not go into implementation details of the MicroBlaze-based main CPU as we have mostly used standard peripherals and design tools provided by Xilinx. We note however the MicroBlaze soft-core implementation is customized to fit in a target FPGA location where the amount and length of the static routes across the chip is minimal, thus increasing the allocatability of hardware tasks. While the most important aspects of the main CPU have already been outlined in section 4.2, this section will explain its interaction with the HWuK. A block diagram highlighting the components that play a role in this interaction is shown in Fig. 10. These are the IDB/ODB and memory, through which data is exchanged between hardware and software tasks, and Task BRAM, which is used to exchange control information between SWuK and HWuK.

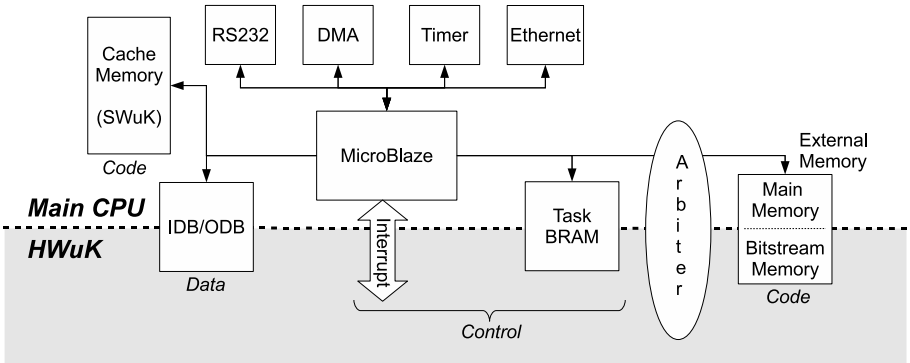


Fig. 10: Block diagram of the main CPU

The data read from the CPU's IDB, which has been generated by hardware tasks, is copied either to the data segment of the corresponding consumer software task in the external memory, or left in the cache for immediate use. Analogously, the data written to the CPU's ODB, which will be delivered to hardware tasks, is retrieved either from the data segment of the corresponding producer software task in the external memory, or directly from the cache memory. An advantage of this scheme is its compatibility with most software compilation systems and most task memory mappings of software OS.

The Task BRAM is used to exchange control information, such as control commands (e.g. task management operation codes) and task parameters (e.g. `TaskID`, data retrieval time, configuration and execution times). Every time this memory is written to by one of the parts (i.e. SWuK or HWuK), an interrupt is provoked in the other part, and when this reads the written data, the interrupt is cleared. Communications can be initiated either by SWuK, when marking a task as ready, or by HWuK, either when indicating that a task has been scheduled and requires input data to be delivered, or when a hardware task has completed its computation and results are ready in CPU's IDB. In the latter two situations, the passed parameter from HWuK is the `TaskID` of the scheduled task or of the task that has completed its computation, respectively.

7.2. The R3TOS Software Microkernel (SWuK)

The R3TOS SWuK is currently based on FreeRTOS, which is an easy-to-use and open source real-time microkernel specifically designed to have a small memory footprint. Indeed, our FreeRTOS porting to R3TOS requires 29.8 KB, thus fitting in only 16 BRAMs. Two features of FreeRTOS are especially attractive to us. First, its high *dependability*, which is supported by the fact that a microkernel derived from it, i.e. SafeRTOS, has been certified for safety-critical applications. Second, its *popularity*, which is confirmed by the 2013 EETimes embedded systems market study, where FreeRTOS came on top in two categories: the kernel currently being used, and the kernel being considered for the next project to develop. This is precisely our objective: provide R3TOS with the most attractive software skin for application developers.

FreeRTOS implements message queues as well as binary, counting and recursive semaphores and mutexes for communication and synchronization between real-time tasks, or between real-time tasks and interrupts. Note that these mechanisms are natively available for software tasks, and must be extended to be used with hardware tasks. While the core part of FreeRTOS has been kept intact in our porting to R3TOS, new ISRs have been programmed to enable communication with the HWuK. Likewise, the scheduler included in the commercial distribution of FreeRTOS has been modified to provide the necessary support for dealing with hardware tasks (e.g. preemption is disabled for hardware tasks). Finally, new functionality has been developed to ease common operations in R3TOS (e.g. DMA transfers between IDB, ODB and external memory). We note that the kernel tick used in the SWuK, which is in the range of milliseconds, is coarser than that in the HWuK. This reflects the low achievable time precision when using software instead of hardware.

The application developer can indistinguishably use software and hardware tasks, as both of them are managed in a uniform way by R3TOS. The major difference between them is that, while the body of a software task includes the computation to be performed, hardware tasks have a "ghost software body" whose main objective is to make them manageable in SWuK. Nevertheless, although the generic software body of a pure hardware task typically includes only HWuK-related system calls, if needed, it could also include other SWuK system calls and regular software code.

In order to reduce time overheads, hardware tasks are assigned the highest priority in SWuK and thus, they are immediately executed by SWuK's scheduler when

they are ready. Immediately after a hardware task is inserted in the ready queue of the Task BRAM (`insert_task()` system call), it is blocked in the software level (`wait_scheduling()` system call) until either the HWuK's scheduler selects it to be executed on the FPGA or it misses its deadline. When the hardware task is scheduled by HWuK scheduler, and once the allocator has found an allocation for it, the task is awakened in the software level. After transferring the input data to be processed by the task to the ODB (`wr_ODB()` system call), the task is again blocked in the software level (`wait_computing()` system call), but starts computing in hardware. When the latter computation finishes, the software task is awakened and retrieves the computed results from the IDB (`rd_IDB()` system call). It is important to note that the aforementioned task blocking and awakening mechanisms in the software level are based on binary semaphores provided by FreeRTOS. The semaphores are taken when executing `wait_scheduling()` or `wait_computing()` system calls, and they are released upon reception of the suitable control commands from HWuK. The interaction between the software and hardware levels during the life cycle of a task is shown in Fig. 11.

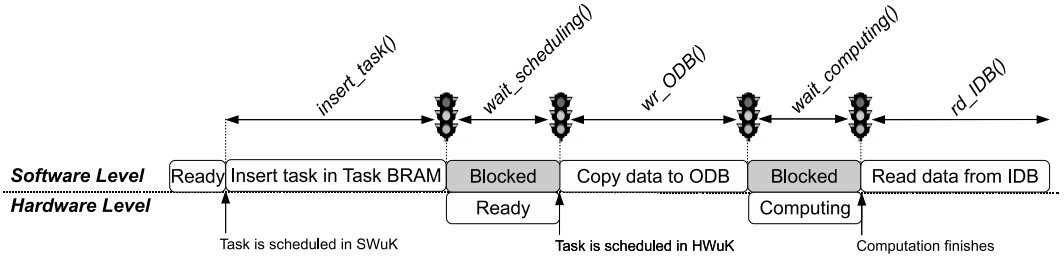


Fig. 11: Management of a hardware task in SWuK

8. PROOF-OF-CONCEPT IMPLEMENTATION OF R3TOS: SDR CASE-STUDY

A R3TOS prototype has been implemented on a Xilinx Virtex-4 XC4VLX160 FPGA. This chip is very interesting as it includes a 28 CLB column wide homogeneous region in the central part with the heterogeneous resources located in the edges; in the leftmost edge there are 3 BRAM columns and 1 DSP48 column, while in the rightmost edge there are 4 BRAM columns.

As shown in Fig. 12, the R3TOS core circuitry is located in the upper-right quadrant of the chip, leaving 3/4 of the FPGA free to allocate the hardware tasks, i.e. Partially Reconfigurable Regions (PRRs). Note that only the clock distribution lines span across the PRRs to reach the regional clocking resources (i.e. BUFRs), which are located in the leftmost and rightmost IOB columns. Next to the BUFRs, and occupying only one CLB column, are the associated diagnostic circuits to detect errors in the clocking resources. The R3TOS core circuitry comprises two different parts: (i) the HWuK, which spans 2 FPGA rows in height, and (ii) the main CPU, which spans 4 FPGA rows. Both R3TOS parts communicate with each other as well as with I/O device pins through a set of Bus Macros (BMs) located in the rightmost 4 CLB columns of the chip. This modular implementation of R3TOS favours adaptability and upgradability, as the main CPU can be replaced without having to modify the HWuK. In addition, the HWuK includes a set of BMs in the leftmost side to access the FPGA's hard primitives which are located in the central part of the chip (i.e. Frame_ECC, ICAP, STARTUP and BUFGCTRLs).

The prototype consumes 4,401 slices and 30 BRAMs in the FPGA. Both the HWuK and main CPU consume a similar amount of resources (HWuK: 2,003 slices and 6

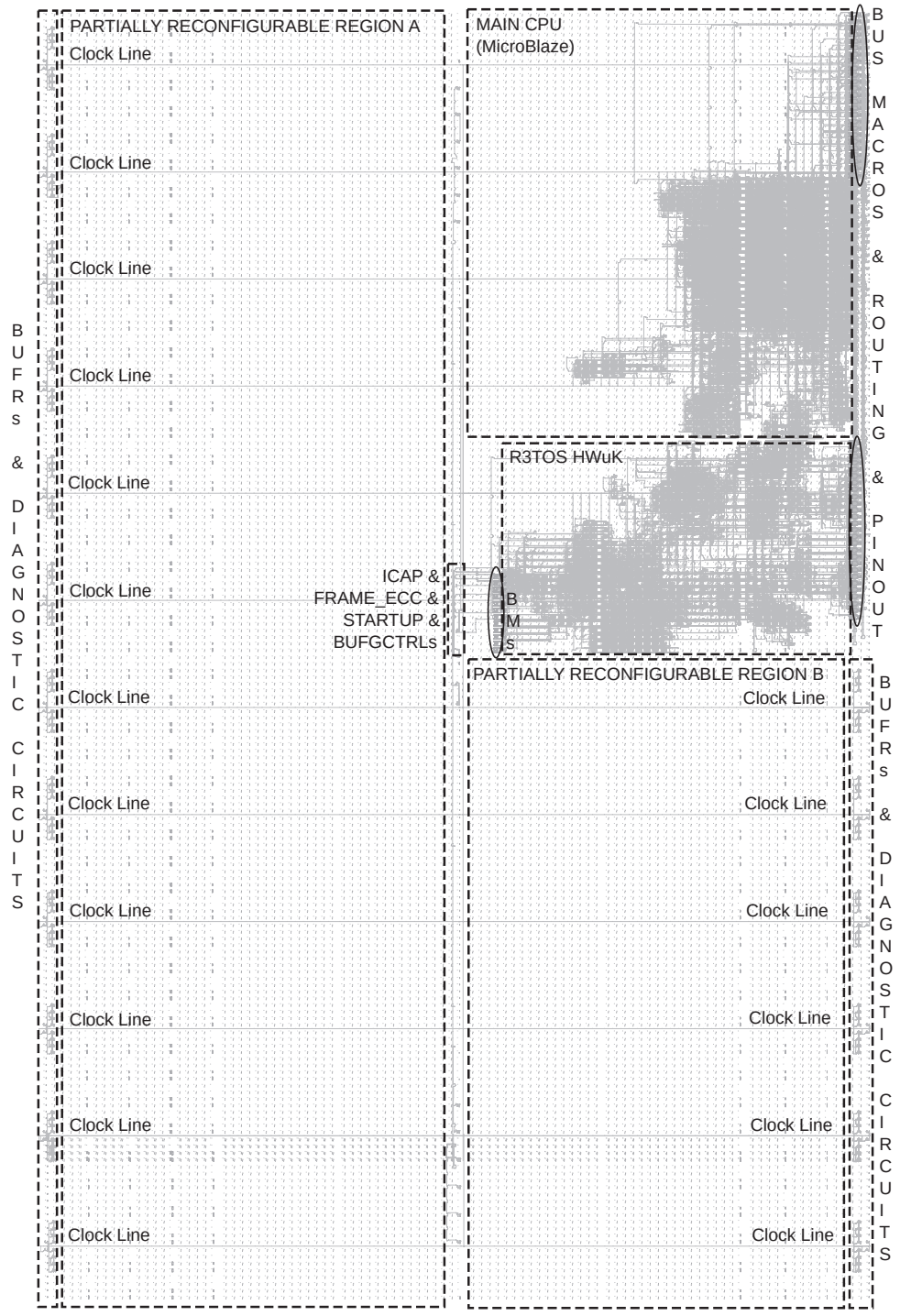


Fig. 12: FPGA implementation of R3TOS proof-of-concept prototype

BRAMs; Main CPU: 2,392 slices and 24 BRAMs), with the configuration manager being the component that requires more logic in comparison with other components (829 slices and 2 BRAMs). We note that the design of this component is not optimized and hence the margin for improvement is still considerable.

8.1. Prototype Functioning

The functioning and performance of the R3TOS prototype has been demonstrated in the context of a SDR application. When used together, R3TOS and SDR permit to build highly reliable systems with the capability to deal not only with "internal" hazards emerging on chip (i.e. spontaneously occurring faults), but also with "external" threats appearing in the environment where the chip is working (e.g. spontaneous interferences). Furthermore, R3TOS permits to use system resources to perform either communication (i.e. SDR), or computation related tasks at different times. Including some computation capabilities in the SDR transmitter (i.e. data pre-processing) allows for significant energy saving and more reduced communication bandwidth as only the significant pieces of information are to be sent. This is especially useful for applications with size, bandwidth, consumption and resource constraints which operate in inaccessible (and harsh) locations, such as remote sensor networks, deep space exploration spacecraft, or offshore wind turbines.

In the SDR case-study, the R3TOS prototype is extended with a superheterodyne Radio Frequency (RF) front-end and a color CIF image sensor. Consequently, the R3TOS implementation shown in Fig. 12 is coupled with some specific static logic to drive the RF front-end (i.e. A/D and D/A controllers), and for receiving the RGB video signal from the CIF sensor to be transmitted. This logic is allocated below the R3TOS core circuitry, in the bottom-right quadrant of the chip. Central to this logic is double buffering, where two memories are alternatively switched between read and write. While one of the memories in the D/A controller is written to by R3TOS, the content of the other memory is transmitted through the RF front-end, and similarly, while one of the memories of the A/D controller is read by R3TOS, the other memory is filled with data received from the RF front-end.

Our SDR prototype implements a simplified version of three of the currently most used communication standards: IEEE 802.11 WiFi, IEEE 802.16 WiMAX, and IMT-2000 UMTS (also known as 3G). Notably, our prototype is able to switch the communication standards on-the-fly to achieve smooth transition of the service as the system moves from one network connection to another. However, as we have not implemented the communication standard detection functionality yet, standard switches are forced by pushing some buttons in the prototype. Additionally, our SDR prototype implements a "home-made" cognitive radio solution which is able to autonomously detect and circumvent any interferences provoked in the transmission channel by using spectral switches. The interferences are generated in selected spectrum bands with an RF white noise source. The implemented modulation in our cognitive radio is Offset Quadrature Phase-Shift Keying (OQPSK). Besides receiving the data to be transmitted, the latter modulator also receives the output Intermediate Frequency (IF), which is computed on-the-fly based on the availability of spectral bands in the transmission channel (it can range between 5 to 10 MHz). Indeed, the Power Spectral Density (PSD) in the transmission channel is periodically evaluated (with a period of 50 ms) using Fourier transform methods to estimate any potential interferences and noise. Refer to [Torrego et al. 2011] for further explanations on this.

The SDR functionality was firstly developed and validated using Xilinx System Generator tool. The natural partitioning of System Generator models into (data-stream processing) tasks consisted in grouping all of the logic in-between consecutive memories together, using the data buffers included in the TCLs to implement the latter

ID	Task Name	Slices	BRAMs	DSP48s
θ_1	JPEG Compressor	2,859	15	2
θ_2	Randomizer	53	10	-
θ_3	Reed-Solomon Encoder	196	10	-
θ_4	Convolutional Encoder	46	10	-
θ_5	Puncturer	39	12	-
θ_6	Inter-leaver	468	12	-
θ_7	Data Symbol Mapper	132	9	-
θ_8	WCDMA Modulator	3,041	8	67
θ_9	OFDM 64-carrier Builder	749	9	-
θ_{10}	Training Sequence Inserter	1,771	18	-
θ_{11}	64-tap IFFT	1,536	11	26
θ_{12}	16-word Cyclic Prefix Inserter	59	11	-
θ_{13}	OFDM 256-carrier Builder	3,332	9	-
θ_{14}	256-tap IFFT	2,585	11	42
θ_{15}	64-word Cyclic Prefix Inserter	903	11	-
θ_{16}	OQPSK Modulator	596	11	70
θ_{17}	PSD Estimator	1,573	11	32

Table II: Task-set in the SDR prototype

memories. However, some pieces of the logic were too large and thus needed to be decomposed into a set of smaller tasks. Namely, the OFDM modulator was divided into three or four smaller tasks: (i) the OFDM symbol builder which allocates the data symbols to orthogonal sub-carriers, (ii) the Inverse Fast Fourier Transform (IFFT) to translate the OFDM symbols to the time domain, (iii) the Cyclic Prefix Inserter (CPI) to mitigate the inter-symbol interference due to multi-path propagation, and in the case of WiFi, (iv) the Training Sequence Inserter (TSI). At this point, it is convenient to clarify that the purpose of this demonstrator is not to come up with the most efficient application partition, but to demonstrate the feasibility of developing R3TOS-based SDR applications. Overall, the task-set listed in Table II was implemented.

The three communication standards implemented by our SDR prototype as well as our cognitive radio solution were built using the aforementioned task-set. The tasks are executed in a different order and with some minor changes in each case. For instance, the generation polynomial used in the convolutional encoder is different in WiMAX and UMTS standards, and the interleaving pattern and length changes from one standard to the other. Therefore, the tasks were parameterized to allow small online adjustments to be made in order to fulfil the requirements of each communication standard. In order to ease this process the adjustable parameters were mapped to directly accessible FPGA resources, such as LUTs. On the other hand, different versions of the same task were developed when the amount of circuitry varies significantly with the values of the adjustable parameters. For instance, θ_{14} is capable of implementing both 64 and 256 IFFT taps, but consumes a notably larger amount of FPGA resources than θ_{11}, which is specifically designed to compute 64-tap IFFT. In order to enable θ_{14}'s reuse to compute 64-tap IFFTs, the size of the IFFT is kept as a parameter in this task. If θ_{14} is not already configured when a 64-tap IFFT is required, the system will proceed to allocate θ_{11}, which takes less time and consumes less FPGA resources.

Xilinx System Generator tool was used to translate the high-level task models into synthesizable VHDL code, which was in turn adapted to the partial reconfiguration design flow, i.e. non-reconfigurable resources were extracted (DCM, BUFG, etc.). The modified VHDL was then combined with R3TOS specific logic (e.g. TCL) and placement constraints (e.g. PRRs definition) prior to synthesizing it to obtain the partial

bitstreams of the tasks. Finally, the latter bitstreams were loaded to the external memory included in the R3TOS prototype.

An important point to be considered in our SDR application is the great difference between the input and output throughput of some of the tasks. For instance, the WCDMA modulator used in cognitive radio (θ_8) generates 1 Kb at its output for each input bit due to the spread spectrum technique used by UMTS. Since the size of the data buffers of the tasks is limited by the availability of BRAMs in the device, tasks with high output-input throughput relation need to execute several times to process all input data, i.e. the remaining input data in the IDB after each task execution must be processed in the next execution.

In general, the execution time of SDR tasks is deterministic. It depends on the amount of data to be processed, the input interface width, and the relation between input and output clock rates. However, the different combinations of variable data produced by the tasks which operate in windows, i.e. the remaining data in the tasks' buffers are accumulated for several executions until they eventually form another data window, results in data transfers of different lengths with the D/A controller. This makes the whole system functioning complex, making it necessary to use *queueing theory* to gain a detailed knowledge of it, which is indeed mandatory to prevent memory overflows in the data buffers.

In connection with the above concept, an important aspect to be considered in order to approach real-time is how to model data transfers between hardware tasks and system input/outputs that are carried out through the ICAP. In order to tackle this issue, we resorted to creating a communication specific task, θ_{18}, which does not consume any on-chip resources and consists only of a configuration phase during which data is transferred (i.e. execution time is equal to zero). θ_{18} is thus considered by the R3TOS scheduler together with the standard SDR tasks.

Since the SDR application is highly sequential, the number of tasks that are ready at the same time is limited and hence, scheduling efficiency is not so important. On the other hand, task allocation is especially important in light of optimizing data transfers among successive SDR tasks and promoting the reuse of already configured circuitry in the chip. In line with these two objectives, data producer and consumer tasks are preferably packaged into the same clock region. The central BRAM columns in the clock region are thus used to directly exchange data between the tasks, while the border BRAM columns are switched between the tasks and DRTs, which move data to contiguous clock regions. As there are 3 BRAM columns in the leftmost FPGA half, a maximum of two tasks can be packaged in the same clock region.

We report the basic functioning of the SDR transmitter prototype. Namely, the prototype was able to transmit data when required, circumventing injected interferences in the transmission channel. The data buffers of the tasks were used as test points to check the correct functioning of our prototype. More specifically, the content of these buffers was read-back and compared against the data produced in the System Generator simulation environment (i.e. MATLAB/Simulink). Besides, we report large time overheads in our prototype due to the fact that the execution phases of most of SDR tasks are shorter than their configuration phases. As a result of these overheads, the time needed to transmit an image ranged from less than a second (in WiFi and WiMAX) up to some seconds (in UMTS and cognitive radio), which is not sufficient for most applications (e.g. real-time video streaming).

As an ending note, we emphasize that this SDR case-study shows the feasibility of using R3TOS for developing reconfigurable applications. However, most of the aspects which play an important role in the achievable performance have been neglected (e.g. codesign and application partition into tasks) and should certainly be revisited in a next stage of research. For instance, SDR hardware tasks could be based on a small

processor (e.g. PicoBlaze) coupled with custom logic instead of relying exclusively on custom hardware to perform the computation. By doing so, a more effective trade-off between resource usage (task size would be smaller) and execution time (it would be longer) could be achieved. As task configuration times would likely be smaller than execution times, the situation where R3TOS is expected to achieve better results could be approached. We believe that, although the execution speed of individual computations would decrease, the overall performance of applications with sufficiently large computing demands could be improved due to better exploitation of the multitasking capabilities delivered by the FPGA, i.e. the device could be kept more occupied with tasks performing active computation for longer periods of time. This scenario is interesting as other functionality could be executed using the saved on-chip resources, enabling more sophisticated applications.

8.2. Performance Figures

Several performance figures were measured on the R3TOS prototype when invoking the system calls from both the HAL and API in the context of the SDR case-study. The most significant ones are shown in Table III. These results were obtained when clocking the prototype at 100 MHz, with the HWuK tick equal to 100 μs and the SWuK tick equal to 1 ms. It is clear to see the achievable efficiency improvement when directly using the HAL instead of the API. In the worst-case, up to 183 μs are required due to API's own operations. This overhead includes: (i) the time needed to perform data transfers to/from data buffers, (ii) the delay introduced by FreeRTOS when processing interruptions, and (iii) communications between the main CPU and the HWuK. It is important to note the achievable acceleration when exchanging data among tasks using DRTs or directly accessing the data in the producer task's ODB. While the time needed to transfer the content of a BRAM-based data buffer using the ICAP is about 60 microseconds, the access to the BRAM can be switched between two tasks within only 10.03 microseconds (around 6x speed-up). In addition, 36.18 microseconds are needed to configure a DRT (around 1.6x speed-up), which then requires 81.92 microseconds to complete the data transfer through the functional layer. Note that the latter time does not constrain the performance as it can be parallelized with the task configuration phase.

9. CONCLUSIONS AND FUTURE WORK

This article has outlined the most important architectural and implementation aspects of R3TOS microkernel and described its functioning principles. The proposed architecture is highly modular and makes extensive use of process parallelism to achieve a good performance, while keeping the area overheads at reasonable bounds. The main focus of the article is on the hardware implementation of R3TOS which is amenable to be used in similar research efforts. This includes: (i) a hardware interface for hardware tasks that makes them self-contained within their boundaries and fully relocatable on the chip (i.e. TCL); (ii) a logic to be attached to the aforementioned task interfaces in order to speed-up the exchange of large amounts of data between tasks (i.e. DRT); (iii) a solution to enable hardware task preemption, (iv) a circuit to ensure task isolation in the configuration domain of the FPGA, and (v) a solution for the Frame_ECC and ICAP coupling problem existing in Virtex-4 devices.

Moving up the abstraction layer, the article presented a set of services which are built upon the microkernel hardware. These exploit the reconfiguration possibilities delivered by Xilinx technology for improving multitasking and dependability. The article also outlined a high-level FreeRTOS-based API intended to universalize the use of reconfigurable hardware and increase productivity.

	HAL		API	
	Min.	*Max.*	*Min.*	*Max.*
TASK MANAGEMENT				
Task execution overhead	-	-	18 μs	18 μs
Read/Write from/to data buffer	-	-	< 1 μs	165 μs
DEALLOCATION				
Deallocation of a hardware task	2.3 μs	33.2 μs	-	-
TASK PREEMPTION				
Disable writing to all flip-flops at start-up	< 2.5 ms	< 2.5 ms	-	-
Context save / restore	26.6 μs	26.6 μs	-	-
INTER-TASK COMMUNICATIONS				
Transfer LUT data buffer (ICAP)	3.7 μs	3.7 μs	-	-
Transfer BRAM data buffer (ICAP)	60.18 μs	60.18 μs	-	-
Transfer BRAM data buffer (DRTs): Conf.	36.18 μs	39.9 μs	-	-
Transfer BRAM data buffer (DRTs): Func.	81.92 μs	81.92 μs	-	-
Switching BRAMs between neighbor tasks	10.03 μs	10.03 μs	-	-
INTER-TASK SYNCHRONIZATION				
Polling of a HWS	1.6 μs	1.6 μs	-	-
Activation of a HWS	3.7 μs	3.7 μs	-	-
CLOCKING MANAGEMENT				
Enable/Disable a BUFR	≈ 4 μs	≈ 4 μs	-	-
Adjust task clock frequency	≈ 4 μs	≈ 4 μs	-	-
RELIABILITY				
Scrub a configuration frame	4.81 μs	9.34 μs	-	-

Table III: Performance figures in the R3TOS prototype

Finally, a R3TOS proof-of-concept prototype has been presented and evaluated in the context of a SDR application. Specifically, the major implementation details and performance measurements have been described.

It is important to note that RT3OS is currently being ported to Xilinx 7-series FPGAs, which include new opportunities for reconfiguration that are envisaged in the future. Xilinx Zynq, with its embedded reconfigurable fabric and built-in ARM processor cores, is another family of devices that we are planning to target in the near future. Besides, we plan to explore other APIs commonly used in high-performance parallel-computing, such as CUDA and OpenCL.

REFERENCES

M. Abramovici, C. E. Stroud, and J. M. Emmert. 2004. Online BIST and BIST-based Diagnosis of FPGA Logic Blocks. *IEEE Transactions on Very Large Scale Integration Systems* 12, 12 (2004), 1284–1294.

A. Ahmadinia, C. Bobda, J. Ding, M. Majer, J. Teich, S.P. Fekete, and J. C. van der Veen. 2005. A Practical Approach for Circuit Routing on Dynamic Reconfigurable Devices. In *Proc. of the IEEE Intl. Workshop on Rapid System Prototyping*. 84–90.

A. Ahmadinia, C. Bobda, D. Koch, M. Majer, and J. Teich. 2004. Task Scheduling for Heterogeneous Reconfigurable Computers. In *Proc. of the Intl. Symposium on Integrated Circuits and System Design*. 22–27.

A. Amouri and M. B. Tahoori. 2011. A Low-Cost Sensor for Aging and Late Transitions Detection in Modern FPGAs. In *Proc. of the Intl. Conference on Field-Programmable Logic and Applications*. 329–335.

D. Andrews, W. Peck, J. Agron, K. Preston, E. Komp, M. Finley, and R. Sass. 2005. hthreads: A Hardware/Software Co-Designed Multithreaded RTOS Kernel. In *Proc. of the IEEE Conference on Emerging Technologies and Factory Automation*.

C. Beckhoff, D. Koch, and J. Torresen. 2012. GoAhead: A Partial Reconfiguration Framework. In *Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines*. 37–44.

N. W. Bergmann, J. A. Williams, J. Han, and Y. Chen. 2006. A Process Model for Hardware Modules in Reconfigurable System-on-Chip. In *Proc. of the Intl. Conference on Architecture of Computing Systems*. 205–214.

B. Blodget, P. James-Roxby, E. Keller, S. McMillan, and P. Sundararajan. 2003. A Self-reconfiguring Platform. In *Field-Programmable Logic and Application (Lecture Notes in Computer Science)*, Vol. 2778. 565–574.

C. Bobda, A. Ahmadinia, M. Majer, J. Teich, S. Fekete, and J. C. van der Veen. 2005. DyNoC: A Dynamic Infrastructure for Communication in Dynamically Reconfigurable Devices. In *Proc. of the Intl. Conference on Field-Programmable Logic and Applications*. 153–158.

G. J. Brebner. 1996. A Virtual Hardware Operating System for the Xilinx XC6200. In *Proc. of the Intl. Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*. 327–336.

K. Chapman. 2010. SEU *Strategies for* V*irtex-5 Devices (*XAPP*864)*. Technical Report. Xilinx Inc.

C. Charmichael and C. W. Tseng. 2009. *Correcting Single-Event Upsets in* V*irtex-4* FPGA *Configuration Memory (*XAPP*1088)*. Technical Report. Xilinx Inc.

J. D. Corbett. 2012. *The* X*ilinx Isolation Design Flow for Fault-Tolerant Systems (*WP*412)*. Technical Report. Xilinx Inc.

S. Corbetta, M. Morandi, M. Novati, M. D. Santambrogio, D. Sciuto, and P. Spoletini. 2009. Internal and External Bitstream Relocation for Partial Dynamic Reconfiguration. *IEEE Transactions on Very Large Scale Integration Systems* 17 (2009), 1650–1654. Issue 11.

A. Donato, F. Ferrandi, M. D. Santambrogio, and D. Sciuto. 2005. Operating System Support for Dynamically Reconfigurable SoC Architectures. In *Proceedings of the IEEE Intl. System-on-Chip Conference*. 233–238.

A. Donlin, P. Lysaght, B. Blodget, and G. Troeger. 2004. A Virtual File System for Dynamically Reconfigurable FPGAs. In *Proc. of the Intl. Conference on Field-Programmable Logic and Applications*. 1127–1129.

S. Dutt, V. Verma, and V. Suthar. 2008. Built-in-Self-Test of FPGAs With Provable Diagnosabilities and High Diagnostic Coverage With Application to Online Testing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27, 2 (2008), 309–326.

A. Ebrahim, K. Benkrid, X. Iturbe, and Chuan Hong. 2012. A Novel High-Performance Fault-Tolerant ICAP Controller. In *Proc. of the NASA/ESA Conference on Adaptive Hardware and Systems*. 259–263.

A. Ebrahim, K. Benkrid, X. Iturbe, and C. Hong. 2013. Multiple-Clone Configuration of Relocatable Partial Bitstreams in Xilinx Virtex FPGAs. In *Proc. of the NASA/ESA Conference on Adaptive Hardware and Systems*.

D. Gohringer, M. Hubner, L. Hugot-Derville, and J. Becker. 2010a. Message Passing Interface support for the Runtime Adaptive Multi-processor System-on-Chip RAMPSoC. In *Proc. of the IEEE Intl. Conference IC-SAMOS*. 357–364.

D. Gohringer, M. Hubner, E. N. Zeutebouo, and J. Becker. 2010b. Operating System for Runtime Reconfigurable Multiprocessor Systems. *Intl. Journal of Reconfigurable Computing* (2010).

S. G. Hansen, D. Koch, and J. Torresen. 2011. High Speed Partial Run-Time Reconfiguration Using Enhanced ICAP Hard Macro. In *Proc. of the IEEE Intl. Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. 174–180.

J. Heiner, B. Sellers, M. Wirthlin, and J. Kalb. 2009. FPGA Partial Reconfiguration via Configuration Scrubbing. In *Proc. of the Intl. Conference on Field-Programmable Logic and Applications*. 99–104.

C. Hong, K. Benkrid, X. Iturbe, A. Ebrahim, and T. Arslan. 2011. Efficient On-Chip Task Scheduler and Allocator for Reconfigurable Operating Systems. *IEEE Embedded Systems Letters* 3, 3 (2011), 85–88.

A. Ismail and L. Shannon. 2011. FUSE: Front-End User Framework for O/S Abstraction of Hardware Accelerators. In *Proc. of the Annual IEEE Intl. Symposium on Field-Programmable Custom Computing Machines*. 170–177.

X. Iturbe, M. Azkarate, I. Martinez, J. Perez, and A. Astarloa. 2009. A Novel SEU, MBU and SHE Handling Strategy for Xilinx Virtex-4 FPGAs. In *Proc. of the Intl. Conference on Field-Programmable Logic and Applications*. 569–573.

X. Iturbe, K. Benkrid, T. Arslan, R. Torrego, and I. Martinez. 2011a. Methods and Mechanisms for Hardware Multitasking: Executing and Synchronizing Fully Relocatable Hardware Tasks in Xilinx FPGAs. In *Proc. of the Intl. Conference on Field-Programmable Logic and Applications*.

X. Iturbe, K. Benkrid, A. Ebrahim, C. Hong, T. Arslan, and I. Martinez. 2011b. Snake: An Efficient Strategy for the Reuse of Circuitry and Partial Computation Results in High-Performance Reconfigurable Computing. In *Proc. of the Intl. Conference on Reconfigurable Computing and FPGAs*.

X. Iturbe, K. Benkrid, C. Hong, A. Ebrahim, T. Arslan, and I. Martinez. 2013a. Runtime Scheduling, Allocation and Execution of Real-Time Hardware Tasks onto Xilinx FPGAs Subject to Fault Occurrence. *Intl. Journal of Reconfigurable Computing* (2013).

X. Iturbe, K. Benkrid, C. Hong, A. Ebrahim, R. Torrego, I. Martinez, T. Arslan, and J. Perez. 2013b. R3TOS: A Novel Reliable Reconfigurable Real-Time Operating System for Highly Adaptive, Efficient and Dependable Computing on FPGAs. *IEEE Trans. Comput.* 62, 8 (2013), 1542–1556.

X. Iturbe, K. Benkrid, R. Torrego, A. Ebrahim, and T. Arslan. 2012. Online Clock Routing in Xilinx FPGAs for High-Performance and Reliability. In *Proc. of the NASA/ESA Conference on Adaptive Hardware and Systems*.

A. Jara-Berrocal and A. Gordon-Ross. 2010. VAPRES: A Virtual Architecture for Partially Reconfigurable Embedded Systems. In *Proc. of the Conference on Design, Automation and Test in Europe*. 837–842.

S. Jovanovic, C. Tanougast, and S. Weber. 2007. A Hardware Preemptive Multitasking Mechanism Based on Scan-path Register Structure for FPGA-based Reconfigurable Systems. In *Proc. of the NASA/ESA Conference on Adaptive Hardware and Systems*. 358–364.

K. Jozwik, H. Tomiyama, S. Honda, and H. Takada. 2010. A Novel Mechanism for Effective Hardware Task Preemption in Dynamically Reconfigurable Systems. In *Proc. of the Intl. Conference on Field-Programmable Logic and Applications*. 352–355.

H. Kalte, G. Lee, M. Porrmann, and U. Ruckert. 2005. REPLICA: A Bitstream Manipulation Filter for Module Relocation in Partial Reconfigurable Systems. In *Proc. of the IEEE Intl. Parallel and Distributed Processing Symposium*.

H. Kalte and M. Porrmann. 2005. Context Saving and Restoring for Multitasking in Reconfigurable Systems. In *Proc. of the Intl. Conference on Field-Programmable Logic and Applications*. 223–228.

H. Kalte and M. Porrmann. 2006. REPLICA2Pro: Task Relocation by Bitstream Manipulation in Virtex-II/Pro FPGAs. In *Proc. of the Conference on Computing Frontiers*. 403–412.

D. Koch, C. Beckhoff, and J. Teich. 2008. ReCoBus-builder - a Novel Tool and Technique to Build Statically and Dynamically Reconfigurable Systems for FPGAs. In *Proc. of the Intl. Conference on Field-Programmable Logic and Applications*. 119–124.

D. Koch, C. Beckhoff, and J. Torresen. 2010. Obstacle-free Two-Dimensional Online-routing for Run-Time Reconfigurable FPGA-based Systems. In *Proc. of the Intl. Conference on Field-Programmable Technology*. 208–215.

D. Koch, C. Haubelt, and J. Teich. 2007. Efficient Hardware Checkpointing: Concepts, Overhead Analysis and Implementation. In *Proc. of the ACM/SIGDA Intl. Symposium on Field-Programmable Gate Arrays*. 188–196.

D. Koch, C. Haubelt, and J. Teich. 2008. Efficient Reconfigurable On-Chip Buses for FPGAs. In *Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines*. 287–290.

H. Kopetz. 2011. *Real-Time Systems: Design Principles for Distributed Embedded Applications* (2nd ed.). Springer-Verlag.

K. Kosciuszkiewicz, F. Morgan, and K. Kepa. 2007. Run-Time Management of Reconfigurable Hardware Tasks Using Embedded Linux. In *Proc. of the Intl. Conference on Field-Programmable Tecnology*.

E. Lubbers. 2010. *Multithreaded Programming and Execution Models for Reconfigurable Hardware*. Ph.D. Dissertation. University of Paderborn, Germany.

J. Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. 2003. Infrastructure for Design and Management of Relocatable Tasks in a Heterogeneous Reconfigurable System-on-Chip. In *Proc. of the Conference on Design, Automation and Test in Europe*.

F. Muller, J. Le Rhun, F. Lemonnier, B. Miramond, and L. Devaux. 2005. A Flexible Operating System for Dynamic Applications. *Xilinx Xcell Journal* Fourth Quarter 2010 (2005), 30–34.

R. Pellizzoni and M. Caccamo. 2006. Adaptive Allocation of Software and Hardware Real-Time Tasks for FPGA-based Embedded Systems. In *Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium*. 208–220.

M. D. Santambrogio, V. Rana, and D. Sciuto. 2008. Operating System Support for Online Partial Dynamic Reconfiguration Management. In *Proc. of the Intl. Conference on Field-Programmable Logic and Applications*. 455–458.

P. Sedcole. 2006. *Reconfigurable Platform-Based Design in* FPGA*s for Video Image Processing*. Ph.D. Dissertation. Imperial College London, UK.

P. Sedcole, P. Y. K. Cheung, G. A. Constantinides, and W. Luk. 2007. Run-time Integration of Reconfigurable Video Processing Systems. *IEEE Transactions on Very Large Scale Integration Systems* 15 (2007), 1003–1016. Issue 9.

M. Shelburne, C. Patterson, P. Athanas, M. Jones, B. Martin, and R. Fong. 2008. Metawire: Using FPGA Configuration Circuitry to Emulate a Network-on-Chip. In *Intl. Conference on Field-Programmable Logic and Applications*. 257–262.

H. Simmler, L. Levinson, and R. Manner. 2000. Multitasking on FPGA Coprocessors. In *Proc. of the Intl. Workshop on Field-Programmable Logic and Applications*. 121–130.

J. Smith, T. Xia, and C. Stroud. 2006. An Automated BIST Architecture for Testing and Diagnosing FPGA Interconnect Faults. *Journal of Electronic Testing: Theory and Applications* 22, 3 (2006), 239–253.

H. K. H. So. 2007. BORPH*: An Operating System for* FPGA*-Based Reconfigurable Computers*. Ph.D. Dissertation. University of California at Berkeley, USA.

A. A. Sohanghpurwala, P. Athanas, T. Frangieh, and A. Wood. 2011. OpenPR: An Open-Source Partial-Reconfiguration Toolkit for Xilinx FPGAs. In *Proc. of the IEEE Intl. Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. 228–235.

M. B. Stensgaard and J. Sparso. 2008. ReNoC: A Network-on-Chip Architecture with Reconfigurable Topology. In *Proc. of the ACM/IEEE Intl. Symposium on Networks-on-Chip*. 55–64.

L. Sterpone and M. Violante. 2005. Analysis of the Robustness of the TMR Architecture in SRAM-based FPGAs. *IEEE Transactions on Nuclear Science* 52, 5 (2005), 1545–1549.

J. Suris, M. Shelburne, C. Patterson, P. Athanas, J. Bowen, T. Dunham, and J. Rice. 2008. Untethered On-The-Fly Radio Assembly With Wires-On-Demand. In *Proc. of the IEEE National Aerospace and Electronics Conference*. 229–233.

R. Torrego, I. Val, and E. Muxika. 2011. QPSK Cognitive Modulator Fully FPGA-implemented via Dynamic Partial Reconfiguration and Rapid Prototyping Tools. In *Proc. of the European Conference on Communications Technologies and Software Defined Radio*.

V. M. Tuan and H. Amano. 2008. A Method for Capturing State Data on Dynamically Reconfigurable Processors. In *Proc. of the Intl. Conference on Engineering of Reconfigurable Systems and Algorithms*.

H. Walder. 2005. *Operating System Design for Partially Reconfigurable Logic Devices*. Ph.D. Dissertation. Swiss Federal Institute of Technology Zurich, Switzerland.

J. A. Williams and N. W. Bergmann. 2004. Embedded Linux as a Platform for Dynamically Self-Reconfiguring Systems-On-Chip. In *Proc. of the Intl. Conference on Engineering of Reconfigurable Systems and Algorithms*.

J. A. Williams, N. W. Bergmann, and X. Xie. 2005. FIFO Communication Models in Operating Systems for Reconfigurable Computing. In *Proc. of the Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. 277–278.

B. Zhou, W. Qiu, and C. Peng. 2005. An Operating System Framework for Reconfigurable Systems. In *Proc. of the Intl. Conference on Computer and Information Technology*. 788–792.

Received February 2007; revised March 2009; accepted June 2009